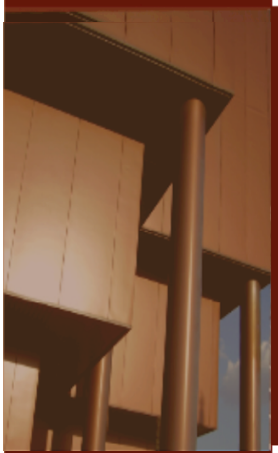
Trabajo de Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Sistema IoT para el control y monitorización de un terrario con Eclipse Kura, Amazon AWS y Angular

Autor: Juan Jiménez Casas
Tutor: María Teresa Ariza Gómez



Departamento de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2019

Trabajo de Fin de Grado
Grado en Ingeniería de Tecnologías de Telecomunicación

Sistema IoT para el control y monitorización de un terrario con Eclipse Kura, Amazon AWS y Angular

Autor:
Juan Jiménez Casas

Tutor:
María Teresa Ariza Gómez
Profesor titular

Departamento de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2019

Trabajo fin de Grado: Sistema IoT para el control y monitorización de un terrario con Eclipse Kura, Amazon AWS y Angular

Autor: Juan Jiménez Casas
Tutor: María Teresa Ariza Gómez

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2019

El Secretario del Tribunal

A mi familia
A mis amigos
A mis maestros

Agradecimientos

Sin duda, el momento más difícil y a la vez más importante en la realización de toda esta memoria no es más que el que acontece en este apartado. Es de especial importancia para mí el acto de agradecer a todas las personas que me han acompañado durante este camino. Por ello, y a pesar de todo, creo que con este apartado nunca será suficiente para agradecer todo lo que os debo.

Agradecer, en primer lugar, a la tutora de este proyecto, María Teresa Ariza Gómez, el tiempo, la confianza y empeño dedicado en ayudar a la realización de este trabajo. Además, me gustaría agradecer al resto de profesores que, en cualquier momento de estos años, me han servido como ayuda y han tendido su mano para que todo esto sea más sencillo.

Todo esto no hubiese sido posible sin vosotros, mis padres, mis hermanos, mis abuelos, mi familia. Muchas gracias, nunca podré compensaros todo lo que hacéis por mí. Este trabajo también es igual de vuestro que mío. Gracias por confiar tanto en mí y por guiarme para ser quien soy. Por enseñarme día tras día, por inculcarme unos valores. Por darlo todo, sin pedir nada a cambio. Os quiero.

No puedo dejar atrás a otro de mis pilares fundamentales, mi compañera en todo este camino, quien me ha apoyado día tras día, ha aguantado mis días malos y me ha ayudado a desconectar en todo el caos que han sido estos cuatro años. Este párrafo va por ti.

Por último, pero no por ello menos importante, me gustaría hacer un agradecimiento a la casualidad, a la casualidad de encontrarme unos compañeros, que pasaron a ser amigos y hoy en día son familia. Sin ellos, hoy en día no estaría escribiendo esto y sin ellos no volvería a repetir. Desde ese primer encuentro casual, las risas por no llorar, los *días tristes* y los *días felices*, los consejos, los momentos vividos, los desvaríos frutos del agotamiento, los agobios y como no, las quejas. No hace falta que os nombre, si ya leyendo este párrafo os habéis sentido identificados, significa que sois parte de esta dedicatoria.

Juan Jiménez Casas
De Las Cabezas de San Juan a Sevilla, 2019

Resumen

Debido a la imparable digitalización que está aconteciendo en la actualidad, numerosos objetos y dispositivos que en el pasado no eran más que eso, se están viendo actualizados y conectados en una red formada por estos. Surge así el concepto de *Internet of Things.*

Este concepto intenta proponer soluciones que hagan que la vida cotidiana de las personas sea mucho más sencilla, haciendo uso de la tecnología. Para ello, se le dota a los objetos de una conectividad que antes no tenían, abriendo paso a un conjunto de infinitas posibilidades, como la facilidad de realizar alguna acción o el análisis de datos para poder tomar mejores decisiones. Como consecuencia, numerosos dispositivos se están viendo conectados y surgen conceptos como las *Smart Cities* o la *Domótica.*

Este trabajo viene a plantear una solución en este campo, haciendo uso de tecnologías de vanguardia, como el uso de Eclipse Kura como *Gateway IoT*[1] en una Raspberry Pi. Además, se ponen en marcha servicios en la nube haciendo uso del conjunto de utilidades que proporciona Amazon AWS, como *IoT Core, DynamoDB, Lambda y API Gateway*, otorgando fiabilidad, seguridad y facilidad de uso al sistema. Para la presentación e interacción con estos sistemas a través de una interfaz web se hace uso de Angular.

[1] Un Gateway IoT es un dispositivo físico o un programa de software que sirve como punto de conexión entre la nube y los controladores, sensores y dispositivos inteligentes.

Abstract

Due to the unstoppable digitalization that is happening today, several objects and devices that in the past were nothing more than that, are being updated and connected in a network formed by them. So, it arises the concept of Internet of Things.

This concept tries to propose solutions that make people's daily life much simpler and easier, making use of technology. To do this, the objects are provided with a connectivity that they did not have before, opening the way to a set of infinite possibilities, such as the ease of performing some action or the analysis of data in order to make better decisions. As a result, several devices are being connected and concepts such as Smart Cities or Home Automation arise.

This work comes to propose a solution in this field, making use of cutting-edge technologies, such as the use of Eclipse Kura as Gateway IoT in a Raspberry Pi. In addition, cloud services are deployed using the set of utilities provided by Amazon AWS, such as IoT Core, DynamoDB, Lambda and API Gateway, providing reliability, security and ease of use to the system. Angular is used to present and interact with these systems through a web interface.

Índice

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

Be the change that you wish to see in the world

- Mahatma Gandhi -

1.1 Motivación

Este proyecto se ha realizado con la finalidad de proponer una solución a un problema que se puede dar tanto en la vida cotidiana de cualquier persona, como en una empresa que tenga una necesidad afín. Para ello se pretende brindar la posibilidad de tener monitorizado un espacio determinado, haciendo uso de sensores de temperatura y humedad principalmente.

Particularmente, surge de la necesidad de tener monitorizado y controlado un terrario[2] de animales, ya que hay ciertos animales, que necesitan tener unas condiciones climatológicas muy controladas. Con este sistema se tendrá un histórico de la temperatura y humedad que ha tenido dicho terrario, haciendo uso de sensores capaces de captar esta información, y una aplicación Web para poder interactuar con el sistema.

Además, se plantea como una solución a negocios que puedan contener un gran número de terrarios, como tiendas de animales, protectoras, zoológicos e incluso invernaderos o botánicas en el ámbito de la agricultura. Por lo tanto, deberá permitir tener estas funcionalidades disponibles en todo momento, con posibilidad de interaccionar desde cualquier lugar, así como una gran escalabilidad.

Para desarrollarlo, se han utilizado una serie de tecnologías, con objeto también de indagar y aprender a utilizar las distintas posibilidades que ofrecen. Una de ellas es Eclipse Kura, un contenedor de aplicaciones Java, que nos permite convertir un dispositivo en un *Gateway IoT*, en el que la importancia está en la reutilización. Además, se hace uso de todo el entorno de Cloud Computing que ofrece Amazon Web Services, desarrollando un sistema *serverless*, de gran utilidad para cumplir con los requisitos de disponibilidad y escalabilidad.

Otra de las tecnologías utilizadas es Angular, un *framework* para aplicaciones web, mantenido por Google, que se utiliza para crear y mantener aplicaciones web de una sola página. Su objetivo es aumentar las aplicaciones basadas en navegador con capacidad de Modelo Vista Controlador, en un esfuerzo para hacer que el desarrollo y las pruebas sean más fáciles.

[2] Instalación en la cual se mantienen artificialmente las condiciones de hábitat adecuadas para ciertos animales de tierra, especialmente reptiles y anfibios, o plantas

1.2 Objetivos

El objetivo del Trabajo es la implementación de un sistema formado por:

- Sensores de temperatura y humedad conectados a una Raspberry Pi, la cual convertiremos en un dispositivo IoT, haciendo uso de Eclipse Kura.
- Entorno de Cloud Computing el cual recibirá información de un dispositivo IoT mediante MQTT, almacenará dicha información en una Base de Datos y la ofrecerá a través de una API REST.
- Aplicación web desarrollada en Angular con la que el usuario pueda obtener información de un determinado terrario y pueda interactuar con este.

1.3 Antecedentes

Los antecedentes de este trabajo corresponden a un proyecto realizado en la asignatura Sistemas Operativos, la cual imparte la tutora de este Trabajo Fin de Grado, María Teresa Ariza Gómez. Aquel proyecto tenía como funcionalidad la de controlar y monitorizar un terrario a pequeña escala, desde la terminal de la Raspberry Pi y haciendo uso de los contenidos impartidos en aquella asignatura, para obtener la información de los distintos sensores.

Como propuesta de mejora a este, se ha buscado el convertir la Raspberry Pi en un dispositivo IoT, para enviar esta información a la nube, con el fin de procesar esta, almacenarla en una base de datos y ofrecerla a través de un servicio REST. Además, se ha desarrollado una aplicación Web que nos permita acceder a la información de este terrario, así como de más terrarios si fuese necesario, con el uso de este servicio.

1.4 Descripción de la solución

A continuación, se describe de manera generalizada la solución implementada, con el fin de alcanzar los objetivos propuestos.

1.4.1 Objetivos Específicos

- **Obtención y envío de los datos de los sensores.**

Para obtener la información de humedad y temperatura se han utilizado sensores DHT11, de bajo coste, pero de gran utilidad para representar la toma de datos. Para la obtención y envío de la información de estos, se ha utilizado Eclipse Kura, que se encarga de obtener la información de los sensores y publicarlas en el IoT Core de AWS, haciendo uso del protocolo MQTT.

- **Publicación de forma segura de información en Amazon AWS.**

Para ello se han utilizado certificados de clave pública y privada, firmados por la propia CA[3] raíz de Amazon. De esta forma, cuando un dispositivo quiera publicar o suscribirse a la información del IoT Core de Amazon, deberá tener un certificado válido.

- **Almacenamiento y consulta de los datos.**

Se ha utilizado una Base de Datos noSQL, llamada Amazon DynamoDB, donde se almacena la información de cada uno de los estados en los que está un Objeto IoT.

[3] Autoridad de certificación.

- **Creación de un servicio serverless, basado en el procesamiento en la nube.**

Haciendo uso de las distintas herramientas que proporciona Amazon Web Services, como son IoT Core, DynamoDB, Lambda Function y API Gateway; se ha implementado un servicio REST sin tener que implementar de forma local un servidor que ofrezca de estos recursos para que luego, una aplicación Web consuma de estos.

- **Objetivos de disponibilidad, escalabilidad y eficiencia.**

Al hacer uso de la computación en la nube, otorgamos a nuestro sistema de una robustez, que, desarrollándolo en un servidor local, no podríamos. Además, con herramientas como AWS Lambda, nuestro servicio solo consume la capacidad de procesamiento que necesita. Esto en conjunción con IoT Core, le otorga una gran capacidad de escalabilidad, por lo que nuestro sistema puede pasar de forma automática de escasas peticiones a un gran número de estas.

- **Presentación**

Para que nuestro sistema pueda ser usable por cualquier usuario se hace uso de una aplicación Web. Esta está desarrollada en Angular. Este *framework* proporciona, entre otras, la capacidad de recarga automática del contenido que se está mostrando en una página, así como la posibilidad de interactuar con una API REST de forma sencilla, haciendo uso del protocolo HTTP.

1.4.2 Funcionalidades

De cara al usuario final, el sistema diseñado ofrece las siguientes funcionalidades:

- Consulta de los distintos terrarios que hay disponibles para consultar.
- Obtención de información de humedad y temperatura de cada uno de los terrarios.
- Modificación del nombre de un terrario, así como de su descripción.
- Acceso al histórico de humedad y temperatura de un terrario.
- Interacción con el sistema IoT, pudiendo activar o desactivar posibles funcionalidades de este, como serían un radiador o una bombilla.

1.4.3 Esquema de la arquitectura

A continuación, se muestra la arquitectura de la solución adoptada:

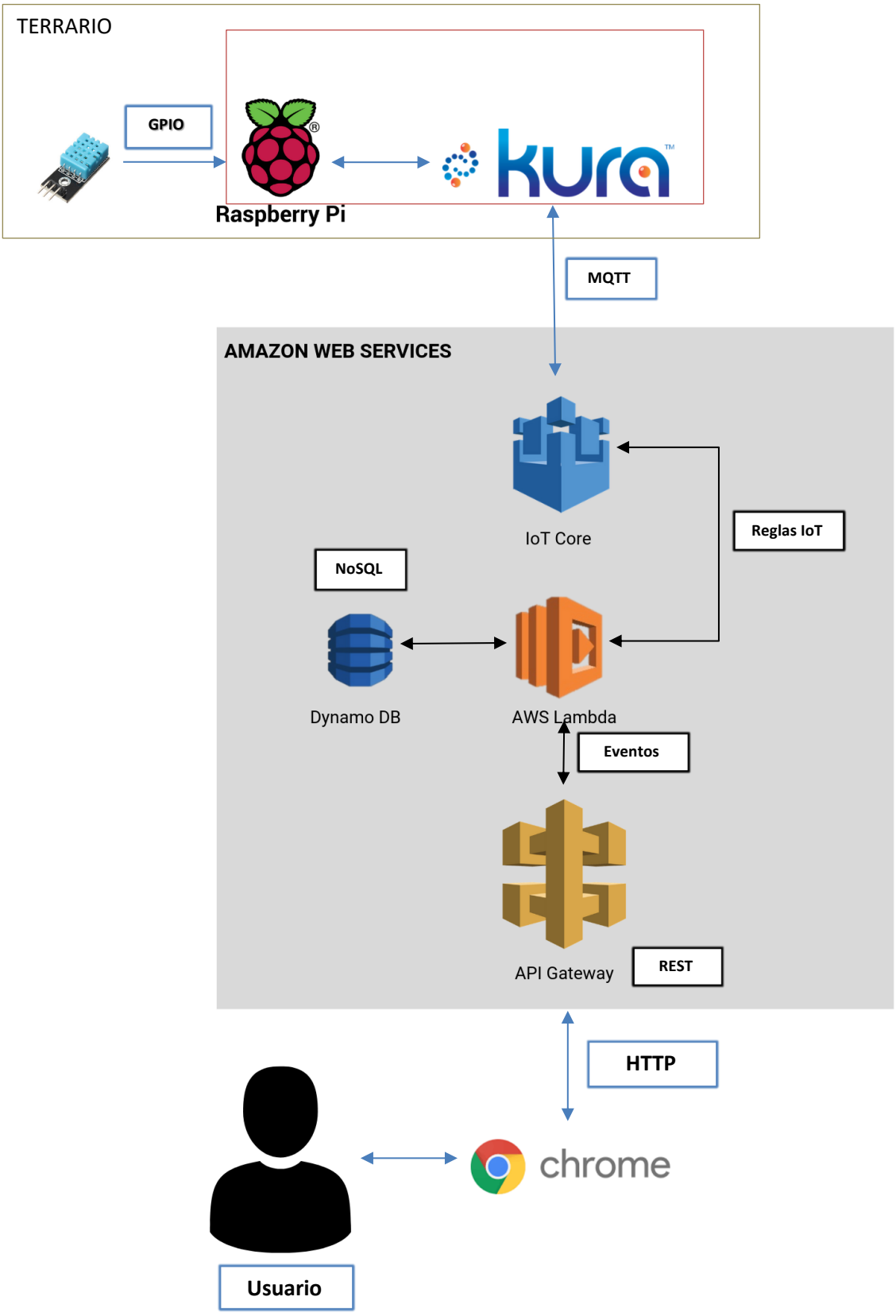


Figura 1: Esquema de la arquitectura

Como podemos observar en la Figura 1: Esquema de la arquitectura, en el escenario formado, un terrario a controlar y monitorizar estaría compuesto por un sensor DHT11 y una Raspberry Pi.

Esta Raspberry Pi tendría instalado Eclipse Kura, que sería el encargado de obtener la información que los sensores DHT11 envían a la Raspberry Pi, mediante los pines GPIO. Una vez tiene esta información, haciendo uso del protocolo MQTT, publicará en un topic MQTT proporcionado por el IoT Core de Amazon AWS. De esta forma, la Raspberry Pi se convierte en un dispositivo IoT. De igual manera, se suscribirá a otro topic para recibir información del IoT Core, enviándosela posteriormente a la Raspberry Pi.

El entorno de Cloud Computing de AWS tendría dos puntos de entrada y salida de información. Uno sería el IoT Core, que mediante el protocolo MQTT se comunicaría con Eclipse Kura y la Raspberry Pi. El otro sería la API REST que proporcionamos haciendo uso del servicio API Gateway, para la comunicación con el usuario, a través de una Aplicación Web Angular.

Internamente, en AWS utilizamos 4 servicios:

- IoT Core: mediante el cual recibimos/enviamos información perteneciente a los Dispositivos IoT, haciendo uso de MQTT.
- API Gateway: proporciona una URL a la que podremos hacer peticiones HTTP, para acceder a los recursos de Amazon AWS. Proporcionaría un servicio REST.
- DynamoDB: proporciona una Base de Datos donde almacenaremos el histórico de humedades y temperaturas que ha tenido un Dispositivo IoT.
- Lambda Functions: funciones ejecutadas cuando se produce un evento o se activa una Regla IoT.

Son el punto de interconexión entre los distintos servicios de AWS. Particularmente:

- Almacenará la información en DynamoDB, cuando se produzcan eventos procedentes de la API Gateway de escritura de información (método POST de HTTP) o cuando se actualice el estado de un Dispositivo IoT en el IoT Core.
- Proporcionará la información almacenada en DynamoDB cuando un evento procedente de API Gateway se lo solicite mediante un método GET de HTTP.

En su conjunto, se forma un sistema interconectado de extremo a extremo, que hace uso de diferentes protocolos, como MQTT o HTTP en cada uno de sus puntos de interconexión. Como resultado, se proporciona la monitorización y control de un terrario.

1.5 Estructura de la memoria

Con el fin de facilitar la lectura del documento, se proporciona en este apartado un resumen de cada capítulo:

1. **Introducción:** Presentación del problema, el contexto que lo involucra y la solución adoptada.

2. **Recursos utilizados:** Recursos hardware y software utilizados para la lectura de la información, despliegue de las distintas tecnologías empleadas (bases de datos, aplicación web, servicios en la nube) así como para implementar los sensores y su comunicación con el sistema.

3. **Tecnologías utilizadas:** Resumen de las principales tecnologías utilizadas para la implementación.

4. **Aplicación desarrollada: Raspberry Pi y Eclipse Kura:** Análisis y Características de la implementación en la Raspberry Pi, así como la publicación de la información en la nube de Amazon.

5. **Aplicación desarrollada: Nube Amazon AWS:** Estructura y análisis de la implementación en Amazon AWS (IoT Core, DynamoDB, Lambda, API Gateway).

6. **Aplicación desarrollada: Interfaz web Angular:** Estructura y análisis de la aplicación web desarrollada en Angular, así como la obtención de la información de la API REST de AWS.

7. **Conclusiones y líneas futuras:** Ideas y características que podrían implementarse para mejorar el sistema en próximas versiones de este.

Además de los capítulos mencionados, se proporciona una serie de Anexos que permitirán implementar el Sistema paso a paso:

- Anexo A: Instalación de Sensores DHT11 y Leds en Raspberry Pi.
- Anexo B: Instalación y configuración de Eclipse Kura en Raspberry Pi.
- Anexo C: Creación de usuarios y configuración de servicios en AWS.
- Anexo D: Despliegue del sistema.

2 RECURSOS UTILIZADOS

The pessimist sees difficulty in every opportunity. The optimist sees the opportunity in every difficulty.

- Winston Churchill -

A continuación se exponen los recursos utilizados para llevar a cabo la implementación. Se detallarán las características más relevantes de cada uno de ellos.

2.1 Recursos Hardware

2.1.1 MacBook Pro

El ordenador en el que se ha desarrollado todo el código necesario para que funcione el trabajo ha sido un MacBook Pro-Retina de 13 pulgadas, de principios de 2015. Entre sus especificaciones encontramos:

- Procesador Intel Core i5 a 2,7 GHz.
- 16 GB de memoria RAM DDR3 a 1867 MHz.
- Gráficos Intel Iris Graphics 6100.
- 128 GB de almacenamiento SSD.

Figura 2: MacBook Pro de 13 pulgadas modelo 2015.

2.1.2 Raspberry Pi Model 3B+

Este es el núcleo de procesamiento y comunicación IoT. En esta se conectarán los sensores de los que se obtendrá la información, los leds para indicar que se está actuando en base a las decisiones tomadas. También será el dispositivo encargado de publicar y suscribirse en la nube para el intercambio de información. Por lo tanto, cumple las funciones de *Gateway IoT.*

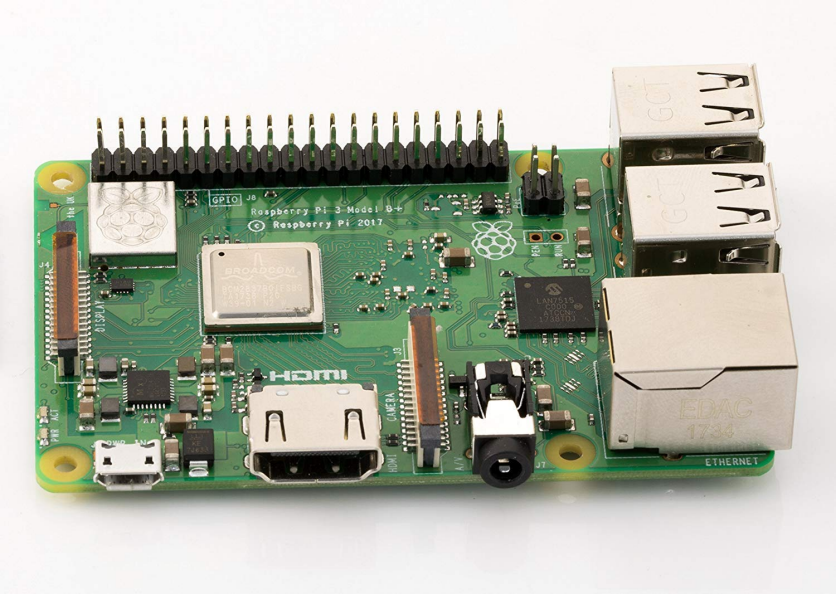


Figura 3: Raspberry Pi Model 3B+

Especificaciones [1]:

- SoC: Broadcom BCM2837B0 quad-core A53 (ARMv8) 64-bit @ 1.4GHz
- GPU: Broadcom Videocore-IV
- RAM: 1GB LPDDR2 SDRAM
- Tarjeta de red: Gigabit Ethernet (via USB channel), 2.4GHz and 5GHz 802.11b/g/n/ac Wi-Fi
- Bluetooth: Bluetooth 4.2, Bluetooth Low Energy (BLE)
- Storage: Micro-SD
- GPIO: 40-pin GPIO header, populated
- Puertos: HDMI, 3.5mm analogue audio-video jack, 4x USB 2.0, Ethernet, Camera Serial Interface (CSI), Display Serial Interface (DSI)

2.1.3 Sensor de humedad y temperatura DHT11

Sensor de humedad y temperatura de bajo coste, muy útil para experimentos y usos donde se necesitan gran cantidad de medidas. La versión utilizada de este es la que contiene 3 pines de conexión con los puertos GPIO de la Raspberry Pi.

Los valores que es capaz de captar son los siguientes [2]:. El implementado es el que se muestra en la Figura 4: Sensor DHT11.

- Humedad: 20-80% (5% precisión)
- Temperatura: 0-50°C (±2°C precisión)

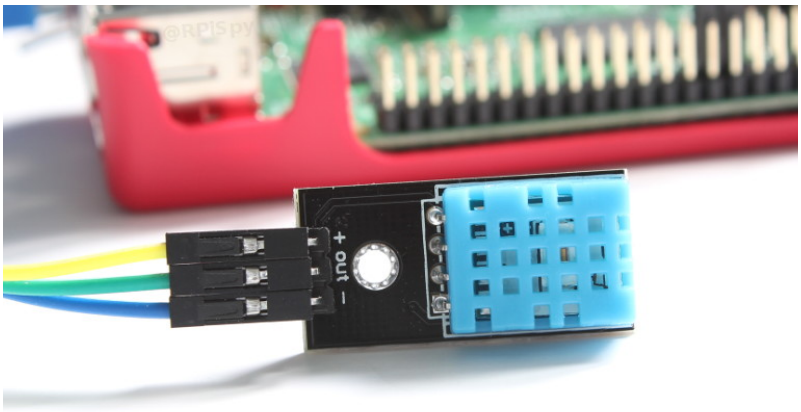


Figura 4: Sensor DHT11

2.1.4 LEDs

Para emitir señales luminosas simulando la actuación sobre cada uno de los posibles elementos con los que interactuar. Los leds son de distintos colores, conectándose estos a la Raspberry Pi a través de una placa de pruebas. Se muestran en la Figura 5: LED

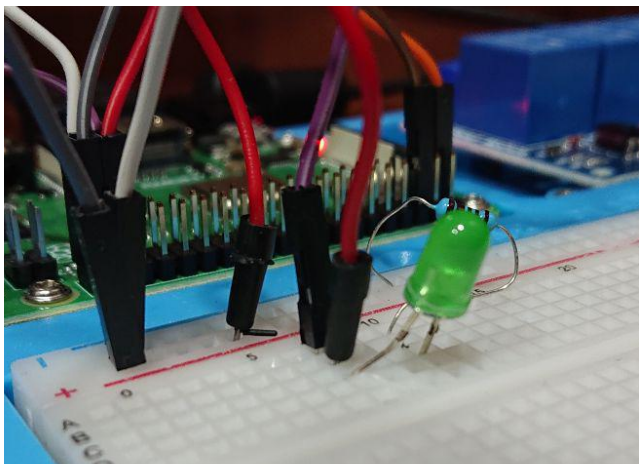


Figura 5: LED

2.2 Recursos Software

2.2.1 Sistema Operativo MacOs Mojave

Este Proyecto se ha desarrollado sobre MacOs Mojave versión 10.14.6, como sistema operativo para los distintos recursos software que han sido necesarios para el desarrollo de la aplicación.

2.2.2 Navegador Web Google Chrome

Se ha utilizado el navegador de Google para la interacción con las distintas interfaces web, así como la depuración y desarrollo de la aplicación web.

Figura 4: Google Chrome

2.2.3 Eclipse IDE

Para el desarrollo de los *bundles* que utilizará Eclipse Kura se ha utilizado el entorno de desarrollo que proporciona Eclipse para Eclipse Kura. [3]

Figura 5: Eclipse

2.2.4 Postman

Es un entorno de desarrollo de API. Ha sido de gran utilidad para desarrollar la API REST, así como verificar la publicación/suscripción a los diferentes temas MQTT.

Figura 6: Postman

2.2.5 StackBlitz y Sublime Text 3

Se han utilizado ambos editores de texto para la codificación. Sublime Text 3 se ha utilizado para desarrollar los controladores de los distintos sensores.
Cabe destacar que StackBlitz ha sido de gran utilidad para el desarrollo de la aplicación Web en Angular, ya que permitía la codificación del código online, guardando automáticamente y un control de versiones en GitHub de forma sencilla. Además, permitirá el despliegue de la aplicación Web mediante un navegador web integrado.

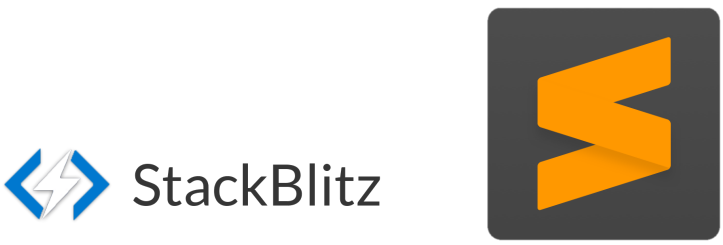


Figura 6: StackBlitz y Sublime Text 3

2.2.6 VNC

Permite utilizar de forma remota la Raspberry Pi. Se accede a esta indicando la dirección IP que tiene asignada e iniciando sesión con el nombre de usuario y contraseña que hayamos configurado.

Figura 7: VNC

3 TECNOLOGÍAS UTILIZADAS

It is easy to hate and it is difficult to love. This is how the whole scheme of things works. All good things are difficult to achieve; and bad things are very easy to get.

- Confucius -

En este capítulo se indican las diferentes tecnologías utilizadas para la implementación del sistema.

3.1 Eclipse Kura

Eclipse Kura [4] es un proyecto llevado a cabo por Eclipse IoT [5] con la finalidad de convertir un dispositivo en un *Gateway IoT* [6]. En este se instalan las distintas aplicaciones haciendo uso de las API que proporciona, permitiendo así despliegue de forma sencilla. Además, proporciona mecanismos de conexión con distintos servicios web, como en nuestro caso es Amazon Web Services.

Figura 8: Eclipse Kura

Un *Gateway IoT* es un dispositivo físico o un programa de software que sirve como punto de conexión entre la nube y los controladores, sensores y dispositivos inteligentes. Todos los datos que se mueven a la nube, o viceversa, pasan por el Gateway. También puede denominarse pasarela inteligente.

Algunos sensores generan decenas de miles de puntos de datos por segundo. Con el Gateway se pre-procesa localmente esta información antes de enviarla a la nube. Minimizándose así el volumen de datos que deben ser enviados a la nube, lo que puede tener un gran impacto en los tiempos de respuesta y en los costes de transmisión de la red.

Con Eclipse Kura se consigue esta funcionalidad, convirtiendo una Raspberry Pi en un Gateway IoT. Para que todo esto tenga una funcionalidad, este además es un contenedor de aplicaciones basado en Java y OSGi [7], denominados bundles, que se desarrollan haciendo uso del entorno de desarrollo Eclipse.

3.1.1 OSGi

El término OSGi generalmente se refiere a cualquiera de las organizaciones de la OSGi Alliance [7]. La tecnología OSGi ofrece un marco orientado a servicios basado en componentes y ofrece formas estandarizadas para gestionar el ciclo de vida del software.
En su creación no se buscaba la posibilidad de ejecutar varias aplicaciones en una máquina virtual única. Los servidores de aplicaciones ya lo hacían (aunque aún no estaban ahí cuando se comenzó, en el año 1998). El problema era más difícil. Se buscaba una forma de reunir diferentes componentes reutilizables que no tenían conocimiento, a priori, de los demás [8].

Aún más difícil, se intentaba que las aplicaciones se construyeran en forma dinámica, utilizando un conjunto de componentes. El objetivo era permitir que nuevas funciones se añadieran a las demás sin necesidad de que los desarrolladores tengan conocimiento íntimo de las demás.

La Figura 9: Arquitectura OSGi muestra la arquitectura OSGi, que tiene fundamentalmente los siguientes componentes:

- **Bundles**: son los componentes que fueron construidos por los desarrolladores, tanto internos como externos. (JAR)
- **Services:** La capa de servicios conecta los bundles de forma dinámica mediante el modelo publicación-búsqueda-suscripción.
- **Life-Cycle:** La API para instalar, iniciar, detener, actualizar y desinstalar los bundles.
- **Modules**: Es la capa que define cómo los bundles importan y exportan código.
- **Security**: La capa que maneja los aspectos de seguridad.
- **Execution Environment**: define qué métodos y clases son accesibles en una plataforma específica para que puedan ser utilizados en una Máquina Virtual Java (JVM).

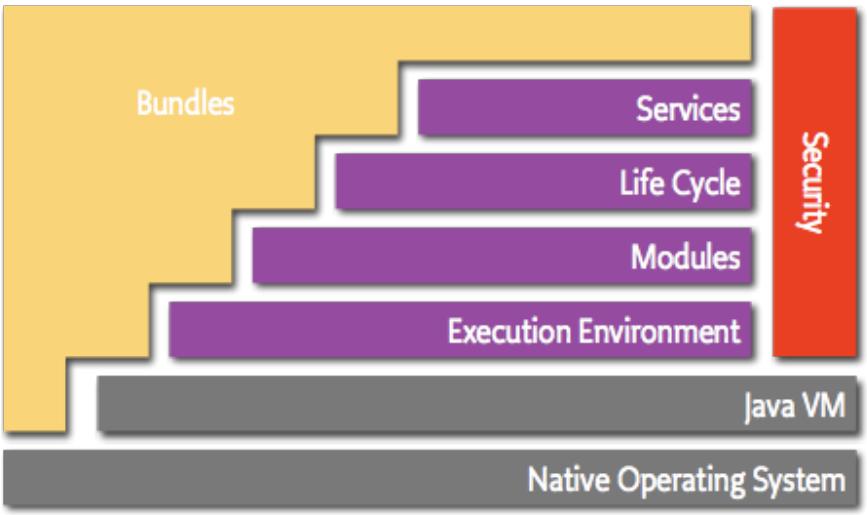


Figura 9: Arquitectura OSGi

Gracias a la ocultación del código, en componentes, la exportación/importación explícita brinda muchos beneficios, como por ejemplo permitir que múltiples versiones del mismo bundle estén activo en un mismo entorno productivo, o la aplicación de seguridad que se hace de forma automática al desplegar estos bundles, entre otros [9].

3.1.1.1 Bundles

Uno de los conceptos claves en OSGi es el “bundle” [10], que es un componente muy parecido a un `.jar` tradicional de Java, con sus interfaces, implementaciones y un archivo manifiesto. En dicho componente se define cierta meta-información que especificará los servicios exportados y los permisos necesarios para poder ser utilizados por otros componentes.

El segundo concepto clave es la administración de los "bundles". OSGi nos permite la gestión del ciclo de vida del software (tales como instalación, parada, redespliegue, arranque o desinstalación de paquetes) chequeando las dependencias existentes para el paquete tratado y evitando conflictos de ejecución. Nos permite también la gestión de versiones de los paquetes, permitiendo la coexistencia de distintas versiones según necesidades y dependencias establecidas, todo esto en tiempo de ejecución.

3.2 Amazon Web Services (AWS)

Amazon Web Services, es la plataforma de servicios en la nube que lanzó la compañía estadounidense de comercio electrónico en el 2006. Esta ofrece servicios de infraestructuras de Tecnologías de la Información (TI) para empresas en forma de lo que hoy conocemos como Cloud Computing[4], facilitando su crecimiento y escalabilidad.

Actualmente, Amazon Web Servicies ofrece una plataforma de infraestructura en la nube de bajo coste [11], llegando a ser gratuita en numerosos casos. Ampliable y de alta fiabilidad, sustenta a cientos de miles de empresas en 190 países del mundo. Clientes de todos los sectores disfrutan de los siguientes beneficios [12]:

- **Bajo coste:** Se puede construir y administrar una infraestructura global a escala y con precios inferiores a los de la competencia, y en algunos casos gratuita.

- **Agilidad y elasticidad instantánea:** Proporciona una infraestructura global y masiva en la nube que permite innovar, experimentar e iterar con rapidez. Se pueden implementar al instante nuevas aplicaciones, desplegando nuevos servidores en segundos, aumentar o reducir la escala al instante conforme a la carga de trabajo.

- **Accesibilidad y flexibilidad:** Es una plataforma independiente del sistema operativo y del lenguaje de programación utilizado. Sólo hay que centrarse en la plataforma de desarrollo o el modelo de programación que más sentido tenga. Esto permite centrarse en la innovación y no en la infraestructura.

- **Seguridad:** es una plataforma tecnológica segura y duradera que cuenta con certificaciones y auditorías reconocidas en el sector: PCI DSS nivel 1, ISO 27001, FISMA Moderate, FedRAMP, HIPAA y SOC 1 (conocida previamente con el nombre SAS 70 y/o SSAE 16) e informes de la auditoría de SOC 2. De esta forma, se otorga al sistema de una seguridad en la nube, que difícilmente se tendría en un sistema creado localmente.

- **Visibilidad de la conformidad:** controlar, auditar y administrar la identidad, la configuración y el uso son elementos cruciales en la actualidad. Con la nube de AWS estas capacidades vienen integradas en la plataforma. Esto, te ayuda a satisfacer los requisitos de conformidad, gobernanza y normativa.

- **Capacidades híbridas:** tiene compatibilidad con el resto de los servicios que se tengan ya desarrollados, por lo que puede existir la coexistencia de estos recursos.

Para llevar a cabo este trabajo, se han utilizado numerosos servicios proporcionados por esta plataforma. Concretamente:

[4] Conjunto de herramientas y servicios que suministran según demanda una serie de recursos informáticos por internet con precios de pago por uso y sin inversión por adelantado en infraestructuras.

3.2.1 AWS IoT Core

AWS IoT proporciona una comunicación bidireccional segura entre los dispositivos conectados a Internet (como sensores, actuadores, microcontroladores integrados o aparatos inteligentes) y la nube de AWS. Esto le permite recopilar datos de telemetría de varios dispositivos, y almacenar y analizar los datos. También puede crear aplicaciones que permitan a sus usuarios controlar estos dispositivos [13].

Figura 10: AWS IoT

Para ello hace uso de los componentes siguientes [14]:

- **Gateway de dispositivos**

Permite a los dispositivos comunicarse de forma segura y eficaz con AWS IoT.

- **Agente de mensajes**

Proporciona un mecanismo seguro para que los dispositivos y las aplicaciones de AWS IoT publiquen y reciban mensajes entre sí. Puede utilizar el protocolo MQTT directamente para publicar/suscribirse o una interfaz HTTP REST para publicar.

- **Motor de reglas**

Proporciona funciones de procesamiento de mensajes y de integración con otros servicios de AWS. Puede utilizar un lenguaje basado en SQL para seleccionar datos de los mensajes, procesar y enviar datos a otros servicios, como Amazon DynamoDB y AWS Lambda. También puede utilizar el agente de mensajes para volver a publicar mensajes para otros suscriptores.

- **Servicio de seguridad e identidad**

Comparte la responsabilidad de la seguridad en la nube de AWS. Los dispositivos deben proteger sus credenciales para enviar datos de forma segura al agente de mensajes. El agente de mensajes y el motor de reglas utilizan las características de seguridad de AWS para enviar datos de forma segura a dispositivos u otros servicios de AWS. Esto se consigue haciendo uso de políticas.

- **Registro**

Organiza los recursos asociados a cada dispositivo en la nube de AWS. Es necesario registrar los dispositivos y asociar hasta tres atributos personalizados a cada uno. También es posible asociar certificados e ID de cliente MQTT a cada dispositivo para poder administrarlos y solucionar los problemas que presenten con mayor facilidad.

- **Sombra del dispositivo**

Documento JSON utilizado para almacenar y recuperar información del estado actual de un dispositivo.

- **Servicio Device Shadow**

Proporciona representaciones persistentes de los dispositivos en la nube de AWS. Es posible publicar información de estado actualizada en una sombra de un dispositivo y este puede sincronizar su estado cuando se conecte. Los dispositivos también pueden publicar su estado actual en una sombra para que lo usen las aplicaciones o los demás dispositivos.

3.2.2 Amazon DynamoDB

Amazon DynamoDB es un servicio que proporciona una base de datos NoSQL totalmente administrad, que ofrece un desempeño rápido y previsible, así como una escalabilidad óptima [15]. Además, DynamoDB ofrece el cifrado en reposo, que elimina la carga y la complejidad operativa que conlleva la protección de información confidencial [16].

Figura 11: Amazon DynamoDB

Se ha elegido DynamoDB frente a una base de datos relacional por la facilidad con la que escala y por la optimización frente a la concurrencia en la escritura de datos. Estos son requisitos fundamentales para nuestro sistema, ya que en un futuro podrían existir numerosos terrarios transmitiendo información que debe ser almacenada en nuestra base de datos.

En DynamoDB, cada tabla contiene una serie de atributos, sin embargo, es obligatorio especificar para la tabla una clave primaria[5]. La clave primaria puede ser de dos tipos:

- **Partition key**: Una clave primaria simple compuesta por un único atributo, llamado partition key.
- **Partition key and sort key**: Una clave primaria compuesta por un par de atributos, el atributo partition key y el atributo sort key.

3.2.3 AWS Lambda

AWS Lambda es un servicio informático que permite ejecutar código sin aprovisionar ni administrar servidores, también conocido como *serverless* [17]. Este ejecuta el código solo cuando es necesario, y se escala de manera automática, pasando de pocas solicitudes al día a miles por segundo [18].

Figura 12: AWS Lambda

[5] Clave que nos permite identificar de forma unívoca un elemento de la tabla.

Con AWS Lambda, se puede ejecutar código para prácticamente cualquier tipo de aplicación o servicio de *backend*[6], y sin que se requiera ningún tipo de administración. Lo único que se tiene que hacer es suministrar el código en uno de los lenguajes que admite AWS Lambda [19]. Luego, es cuestión de conectarlo con otros servicios como DynamoDB o API Gateway, entre otros, para obtener la funcionalidad deseada.

3.2.4 API Gateway

Amazon API Gateway es un servicio de AWS para la creación, publicación, mantenimiento, monitorización y protección de las API REST a cualquier escala. Se puede crear una API que obtenga acceso a AWS o a otros servicios web, así como los datos almacenados en la nube de AWS [20].

Como podemos ver en la Figura 13: Esquema uso API Gateway, diferentes dispositivos como aplicaciones Web o móviles, utilizan API Gateway para conectarse con el resto de los servicios que proporciona Amazon AWS, como AWS Lambda o Amazon DynamoDB.

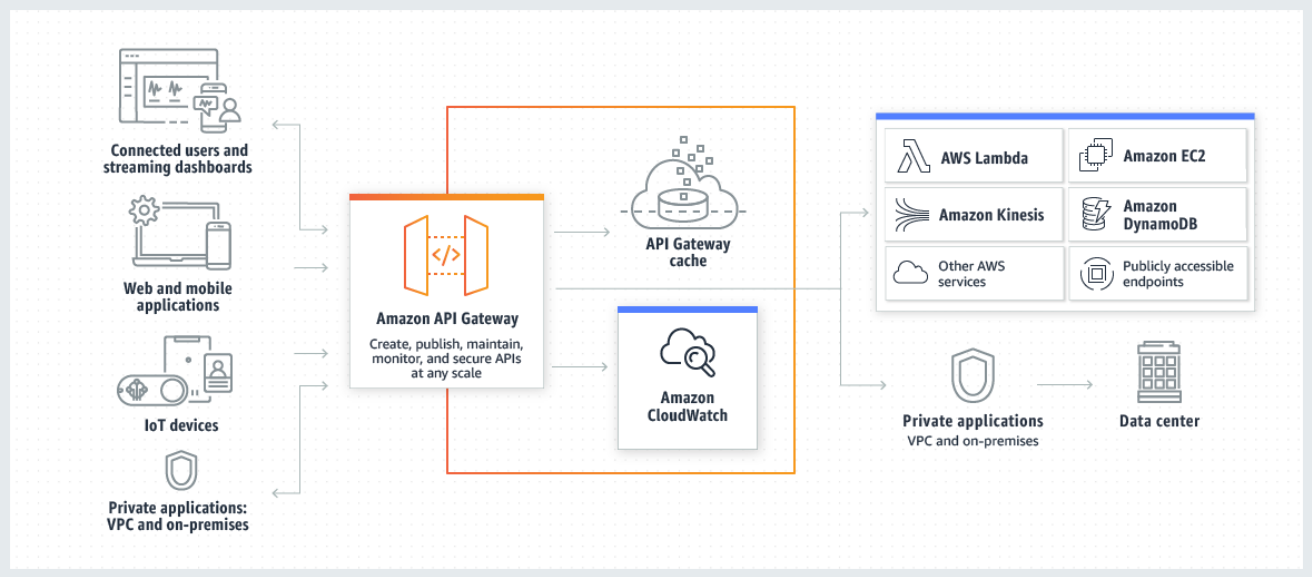


Figura 13: Esquema uso API Gateway

API Gateway crea una API REST que:

- Se basa en HTTP.
- Cumplen el protocolo REST [21], lo que permite la comunicación cliente-servidor sin estado.
- Implementan métodos HTTP estándar como, por ejemplo, GET, POST, PUT, PATCH y DELETE.

Con todo esto, API Gateway será la puerta de enlace de nuestra aplicación Web Angular con el entorno de Cloud Computing de Amazon AWS.

[6] El Backend es la parte trasera de cualquier página web. Se trata de todo el conjunto del desarrollo que se encarga de que una página funcione y de que lo haga como lo hace, pero que al mismo tiempo es totalmente invisible para el usuario, que solo ve lo visual y gráfico.

3.3 Angular

Angular es un *framework*[7] para aplicaciones web desarrollado en TypeScript [22], de código abierto, mantenido por Google, que se utiliza para crear y mantener aplicaciones web de una sola página (SPA). Su objetivo es aumentar las aplicaciones basadas en navegador con capacidad de Modelo Vista Controlador (MVC)[8], en un esfuerzo para hacer que el desarrollo y las pruebas sean más fáciles.

Figura 14: Aplicaciones web de una sola página

Una aplicación web SPA creada con Angular es una web de una sola página, en la cual la navegación entre secciones y páginas de la aplicación, así como la carga de datos, se realiza de manera dinámica, casi instantánea, asíncronamente haciendo llamadas al servidor (*backend* con un API REST) y sobre todo sin refrescar la página en ningún momento.

Angular se basa en clases tipo "Componentes.ts", cuyas propiedades son las usadas para hacer el *binding*[9] de los datos. En dichas clases tenemos propiedades (variables) y métodos (funciones a llamar).
Angular es la evolución de AngularJS [23] aunque incompatible con su predecesor.

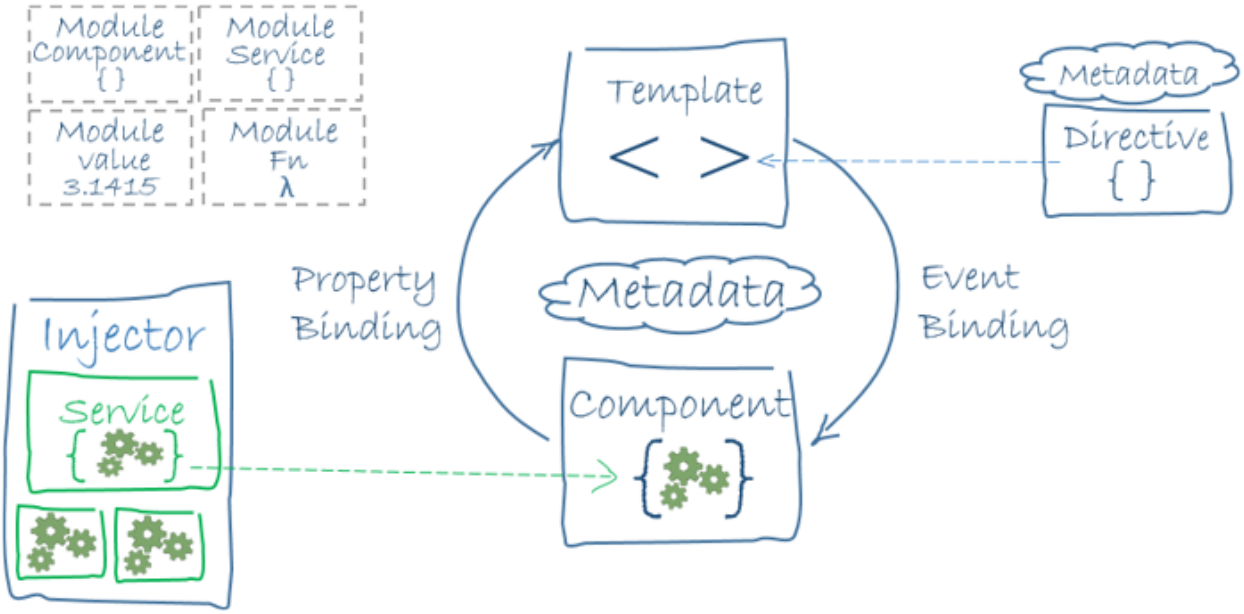


Figura 15: Arquitectura Angular

[7] Un framework es una estructura conceptual y tecnológica de asistencia definida, normalmente, con artefactos o módulos concretos de software, como librerías, que puede servir de base para la organización y desarrollo del software.
[8] Modelo-vista-controlador (MVC) es un patrón de arquitectura de software, que separa los datos y la lógica de negocio de una aplicación de su representación y el módulo encargado de gestionar los eventos y las comunicaciones.
[9] Binding: Enlazado o unión de variables.

A continuación, se explicará la arquitectura que tiene una aplicación en Angular. Para que sea más visual la arquitectura que queremos explicar, cada uno de los apartados que se muestran a continuación hacen referencia a los mostrados en la Figura 15: Arquitectura Angular .

3.3.1 Módulos

El bloque básico para construir una aplicación en Angular son los **NgModules**, estos proporcionan una compilación de los distintos componentes, teniendo en cuenta el contexto y haciendo uso de los módulos que necesite. Será necesario un **módulo root**, el cual nos permitirá indicar el resto de los módulos de los que queremos hacer uso para lanzar la aplicación. Este se llama por convención **AppModule**. Esto ayuda a organizar el código en módulos funcionales, de forma que la administración del desarrollo en aplicaciones complejas sea más sencilla.

3.3.2 Componentes

Cada aplicación en Angular tiene, al menos, un componente: el **componente root**. Este se encarga de conectar los distintos componentes con el contenido que se transfiere a la página Web, conocido como *document object model* (DOM). Cada componente define una clase que contiene los datos de la aplicación y la lógica de esta. Esta clase es asociada con una plantilla HTML, que define la vista en la que se va a mostrar.

Para identificar una clase como componente se utiliza el *decorator*[10] @Component(). Este identifica a la clase en la que se escribe como un componente y le asocia un metadato específico que lo identifica como clase.

3.3.3 Plantillas, directivas y enlace de datos

Las plantillas combinan HTML con marcado en Angular para que los elementos HTML puedan ser modificados antes de ser mostrados. Las directivas de las plantillas proporcionan al programa la lógica y el marcado para el enlazado que conecta los datos de la aplicación con el DOM.

Hay dos tipos de enlazado de datos:

- Event binding: deja a tu aplicación responder a una entrada de información por parte del usuario, actualizando los datos de la aplicación.
- Property binding: deja interpolar los valores que han sido obtenidos desde los datos de la aplicación al HTML.

Antes de que una vista sea presentada, Angular evalúa las directivas y resuelve el enlazado, haciendo uso de las plantillas, para modificar los elementos HTML o el DOM, de acuerdo con la lógica del programa.

3.3.4 Servicios e inyección de dependencias

Para los datos o la lógica que no está asociada con una vista específica y se quiere compartir con los distintos componentes, se crean clases de Servicios. Para la definición de una clase Servicio, se debe acompañar del decorator @Injectable (). Este proporciona los metadatos para que otros componentes puedan inyectar estas dependencias en sus clases.
La inyección de dependencias te permite que tus clases sean limpias y eficientes, ya que no obtienen la información directamente del servidor, sino que delega esto a los servicios, que inyectan lo obtenido.

[10] Un decorator es una función que modifica las clases de JavaScript. Angular tiene definidos una serie de decorators que asocia metadatos específicos a las clases. De esta forma, el sistema sabe qué clases son y cómo deben funcionar.

3.3.5 Routing

El **NgModule Router** de Angular proporciona un servicio que te deja definir una ruta de navegación para los diferentes estados y vistas de la aplicación. De esta forma podemos acceder a cada una de las vistas de la aplicación haciendo uso de la URL en la barra de navegación del buscador o asociando botones de las vistas a rutas.

4 APLICACIÓN DESARROLLADA: RASPBERRY PI Y ECLIPSE KURA

I have not failed. I´ve just found 10,000 ways that won´t work.

- Thomas A. Edison -

La arquitectura del sistema diseñado, consta de tres bloques diferenciados. La implementación en la Raspberry Pi, la parte en la nube de Amazon y la aplicación Web. En este capítulo se profundiza en la implementación que involucra la Raspberry Pi, así como su integración con Eclipse Kura.

4.1 Lectura de información de los sensores en la Raspberry Pi

Los sensores DHT11, se comunican mediante comunicación serie a través de los pines GPIO de la Raspberry Pi. Estos estarán conectados a esta tal y como se indica en el Anexo A: Instalación de Sensores DHT11 y Leds en Raspberry Pi.

A través de estos se puede obtener la información de temperatura y humedad. Para ello, necesitan hacer uso de una librería de Python, llamada **Adafruit**. Esta se encarga de convertir la información recibida mediante comunicación serie a un valor de temperatura y humedad válido. Para la implementación de esta, se ha utilizado Python 3.

El código que es capaz de suministrar esta información se corresponde con el fichero **AdafruitDHT.py**. Este código viene incluido como un enlace, en el paquete que vienen los sensores. Para descargar este se puede acceder a la Web proporcionada como referencia [2].

Para ejecutarlo tenemos que ejecutar el comando que vemos a continuación y pasarle como parámetros el tipo de sensor DHT que es, que en nuestro caso es 11 y el pin GPIO al que está conectado, que en nuestro caso es el 20.

```
$ python AdafruitDHT.py 11 20
```

La información obtenida de esta lectura será escrita en el fichero **temperatura.log** donde se almacenará el histórico de temperatura y humedades. El bundle de Eclipse Kura accederá a este y obtendrá la información de los sensores.

4.2 Publicación y suscripción de la información IoT Core de Amazon AWS. Servicio CloudService.

Para la comunicación con el servicio IoT que proporciona Amazon AWS, utilizaremos el protocolo MQTT. Publicaremos y nos suscribiremos a los topics que el IoT Core nos proporciona para la interacción con las sombras. Los topics se ven en el capítulo 5.1.1 Interacción con el Objeto: Thing Shadow. Eclipse Kura proporciona un servicio para ello, **CloudService** [24]. Este nos proporcionará los servicios que podemos ver en la Figura 16: CloudService.

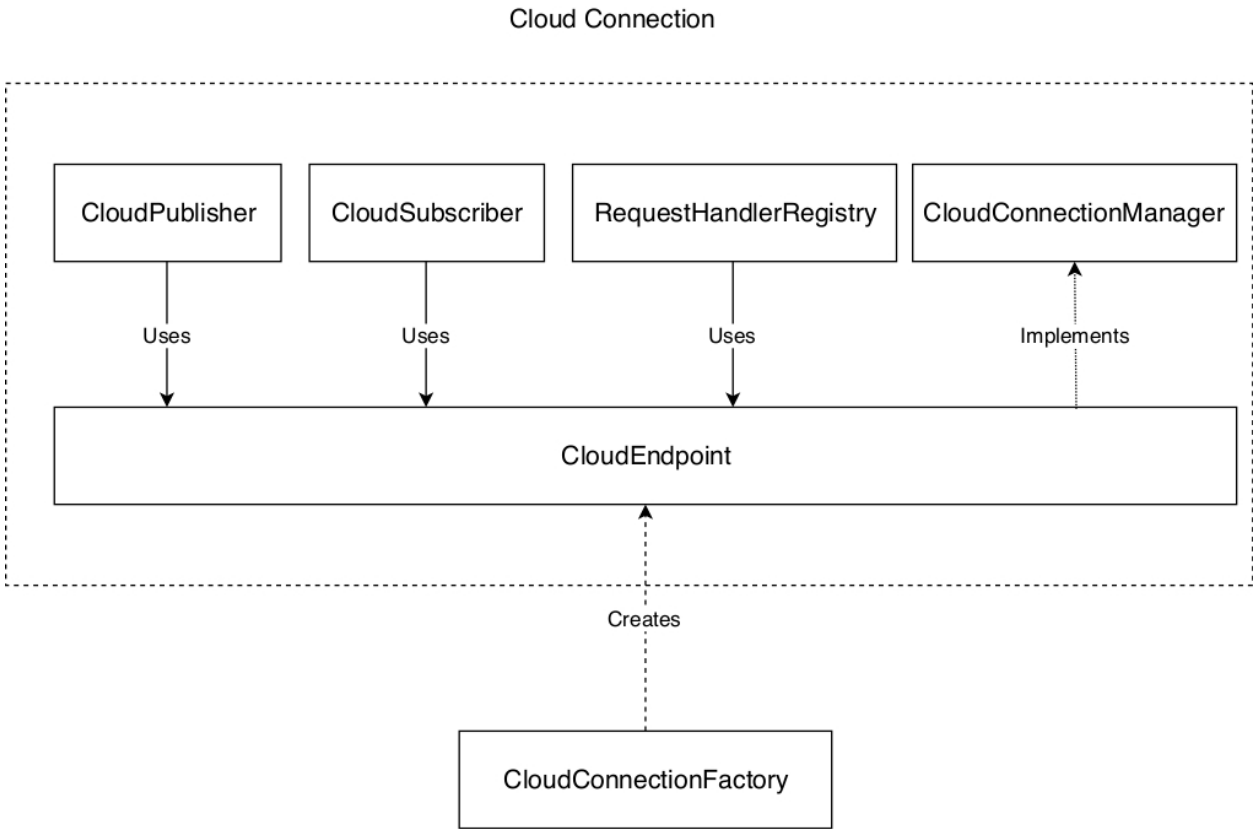


Figura 16: CloudService

Para nuestro sistema, han sido de utilidad los servicios:

- **CloudPublisher**: Permite a las aplicaciones publicar en una plataforma cloud.
- **CloudSuscriber**: Permite a las aplicaciones suscribirse a una plataforma cloud.

Para que todo funcione, es necesario establecer conexión con un **CloudEnpoint**. En nuestro caso será el proporcionado por Amazon AWS. En el apartado Configuración para la compatibilidad con AWS**,** del Anexo B: Instalación y configuración de Eclipse Kura en Raspberry Pi, se muestra cómo crear un CloudService, asociándole un CloudEndpoint [25].

Una vez se ha creado el CloudService, podemos modificar mediante la interfaz Web de Eclipse Kura parámetros de este en el apartado MQTTDataTransport de Cloud Connections, en nuestro caso, asignamos los que se pueden ver en la Figura 17: MQTTDataTransport. Son los siguientes:

- Broker-url: CloudEndpoint proporcionado por Amazon AWS.
- Account-Name: Para una conexión con Amazon AWS. En nuestro caso lo llamaremos `aws`.
- Client-id: Opcional, en nuestro caso le hemos puesto `things`. Lo utilizaremos para establecer la ruta en la que publicar/suscribirnos.
- Topic: Topic en el que queremos publicar. En nuestro caso será $#account-name/#client-id. El "#" lo relaciona con los parámetros (account-name, client-id) mencionados anteriormente.

Con esta configuración, los publicadores/suscriptores lo harán en los topics del CloudEndpoint indicado en el campo Broker-url. Además, lo harán con la ruta `$aws/things` gracias a la información del campo Topic.

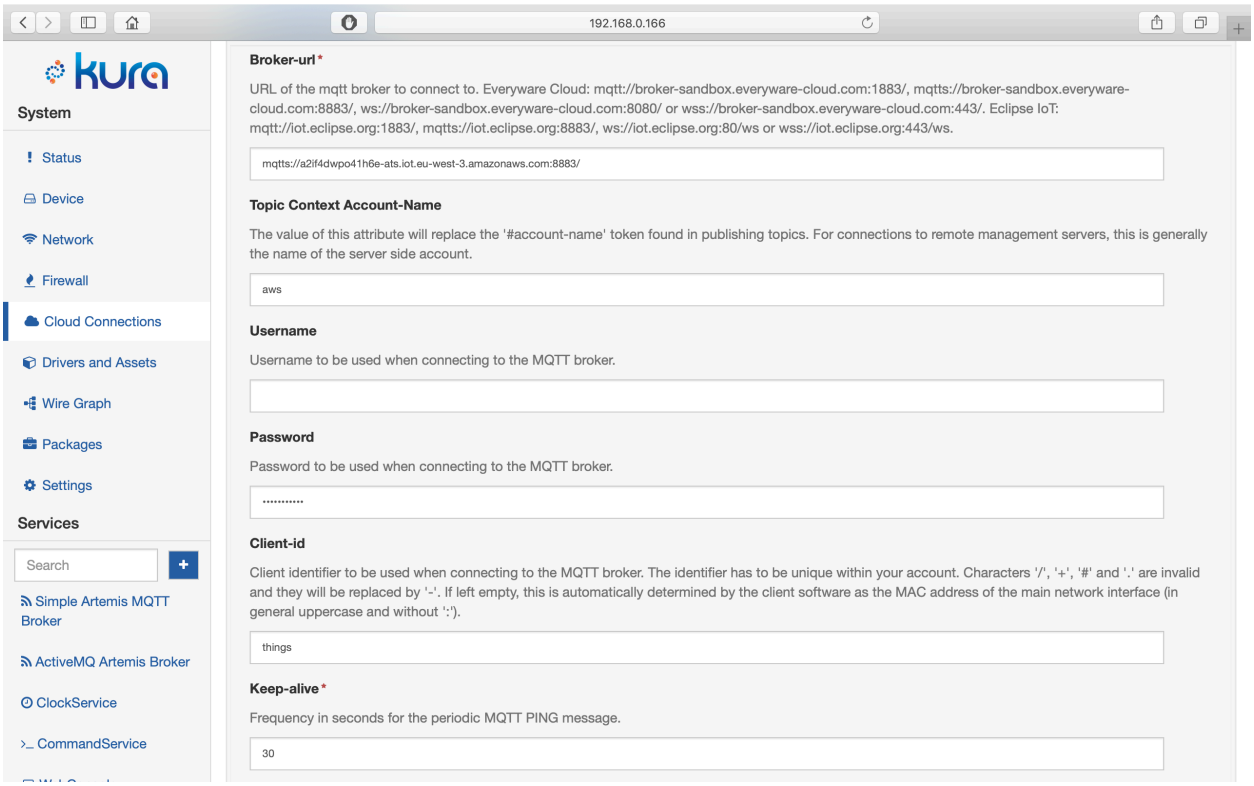


Figura 17: MQTTDataTransport

4.2.1 CloudPublisher

Para publicar la información en un topic de Amazon AWS, crearemos un servicio CloudPublisher. Para crear uno, lo hemos realizado tal y como se indica en el tutorial de Eclipse Kura [25].

Partiendo de que ya hemos creado el CloudService, como se indica en el Anexo B: Instalación y configuración de Eclipse Kura en Raspberry Pi. Debemos ir al menú que aparece en la parte izquierda de la interfaz Web de Eclipse Kura.

En este seleccionamos Cloud Connections. Hacemos click sobre el CloudService que tenemos creado.

Presionamos sobre el botón que indica **New Pub/Sub**. Le asignamos el PID (nombre) que queramos y elegimos que queremos un Publisher. A continuación, se nos mostrará el desplegable de configuración que se muestra en la Figura 18: CloudPublisher.

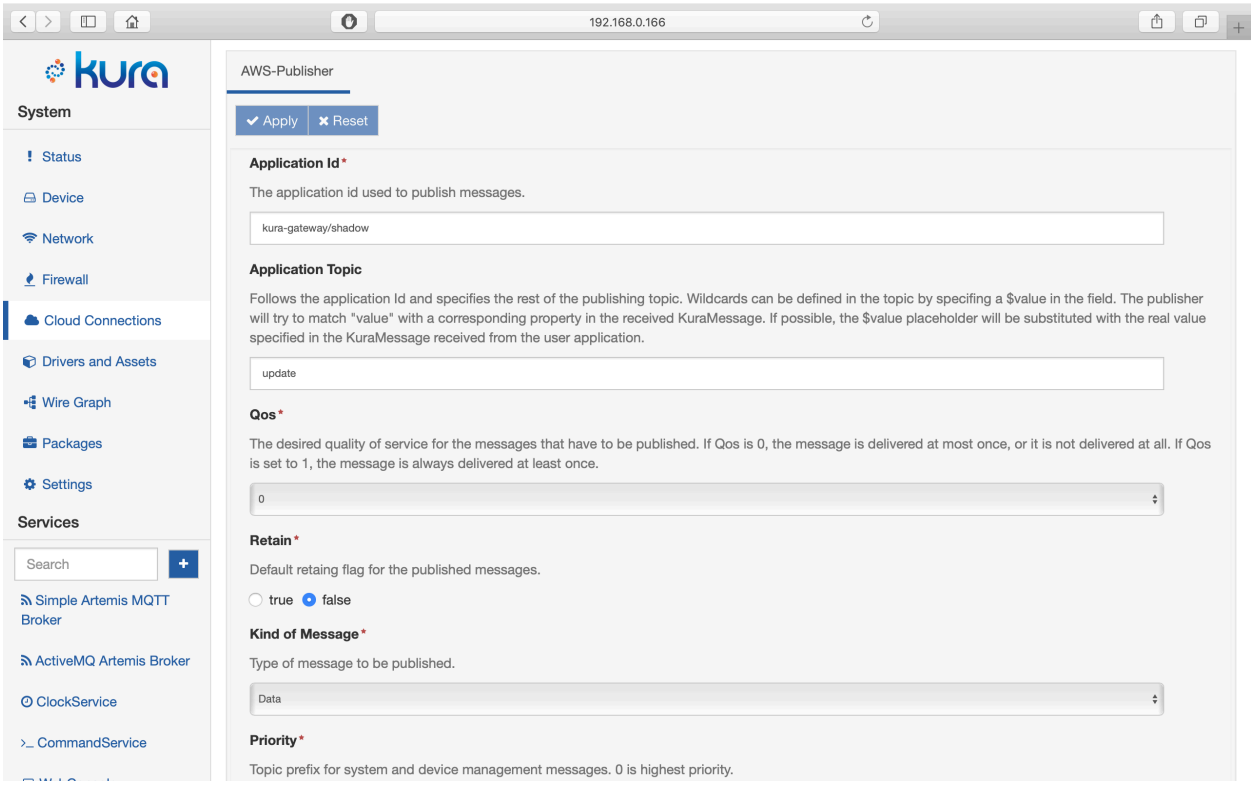


Figura 18: CloudPublisher

Lo destacable de la configuración, que podemos ver en la Figura 18: CloudPublisher es que le hemos indicado en el campo **Application Id**, la ruta para hacer referencia a la sombra de un objeto, en este caso `kura-gateway/shadow`.

En el campo **Application Topic**, le hemos asignado `update`, ya que cuando publicamos, lo que queremos es actualizar una sombra. Por lo tanto, la ruta del topic en el que vamos a publicar es:

```
$aws/things/kura-gateway/shadow/update
```

En el apartado 5.1.1 Interacción con el Objeto: Thing Shadow veremos que se corresponde con la ruta que nos proporciona IoT Core de AWS para actualizar el estado de una sombra mediante MQTT.

4.2.2 CloudSuscriber

La información y configuración en el CloudSuscriber es análoga a la del CloudPublisher. Por lo que para crear un CloudSuscriber, lo podemos hacer como se ha indicado en el apartado 4.2.1 CloudPublisher, pero indicando que queremos un CloudSuscriber en lugar de un CloudPublisher. Lo destacable de este es la ruta del topic a la que nos vamos a suscribir, que es:

```
$aws/things/kura-gateway/shadow/update/accepted
```

Esta es la ruta en la que se publica el estado de una Sombra una vez se ha actualizado.

Para poder ver la información que recibimos de este, lo vamos a enlazar con el servicio **Logger** que proporciona Eclipse Kura. Este servicio nos escribirá en el fichero `/var/log/kura.log` toda la información que recibamos. Para realizar el enlazado, se puede hacer mediante la interfaz Web de Eclipse Kura. Se hace de manera gráfica, tal y como se puede ver en la Figura 19: Kura Wires.

Se deben seleccionar de la columna Wire Components los componentes que queremos enlazar, en este caso **Suscriber** y **Logger**. Con el uso del ratón se conectan ambos componentes.

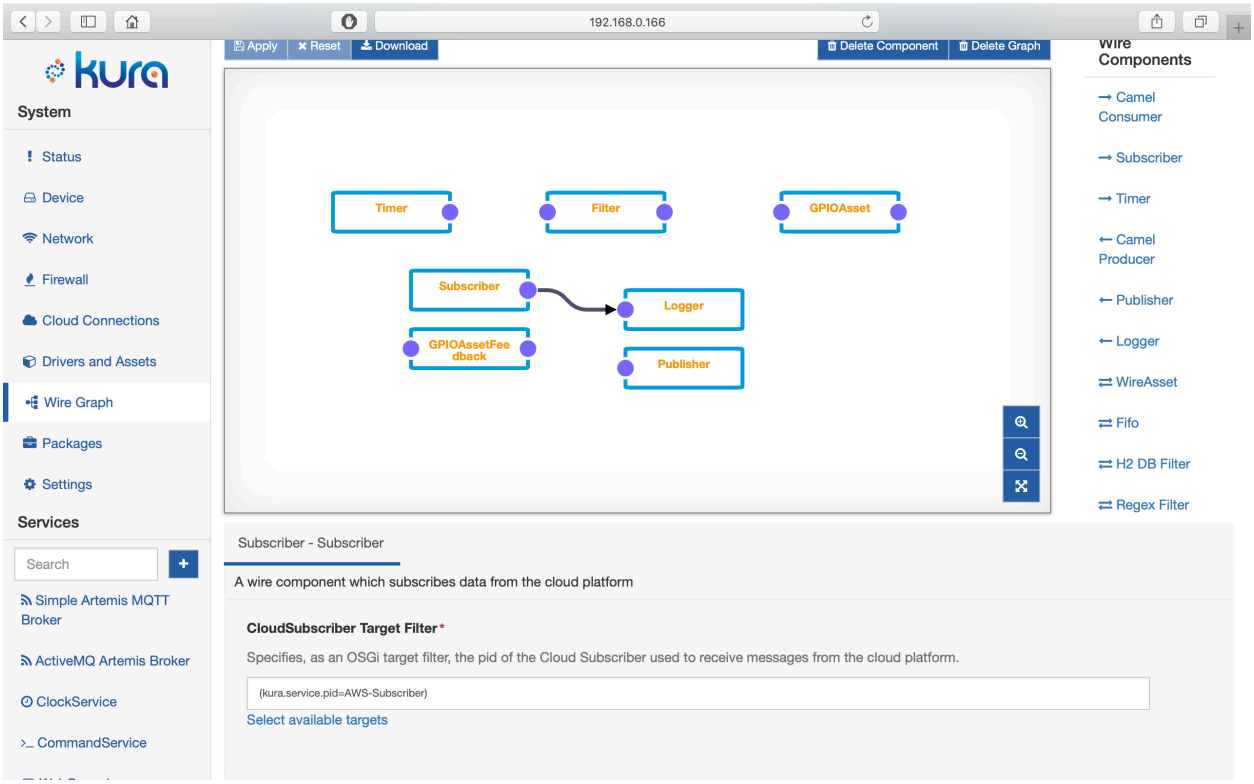


Figura 19: Kura Wires

De esta forma, si mostramos el contenido del fichero `/var/log/kura.log` obtenemos la información recibida del topic al que nos hemos suscrito. Esto se muestra en la Figura 20: Fichero /var/log/kura.log.

```
2019-09-05T17:56:18,742 [] INFO  o.e.k.d.h.Heater - Published message: org.eclipse.kura.message.KuraPayload@371b9e
2019-09-05T17:56:18,745 [DataServiceImpl:Submit] INFO  o.e.k.c.d.t.m.MqttDataTransport - Publishing message on topic: aws/things/kura-gateway/shadow/update with QoS: 0
2019-09-05T17:56:18,787 [MQTT Call: things] INFO  o.e.k.c.c.CloudServiceImpl - Message arrived on topic: aws/things/kura-gateway/shadow/update
2019-09-05T17:56:18,792 [MQTT Call: things] INFO  o.e.k.i.w.l.Logger - Received WireEnvelope from org.eclipse.kura.wire.CloudSubscriber-1567686495335-4
2019-09-05T17:56:18,793 [MQTT Call: things] INFO  o.e.k.i.w.l.Logger - Record List content:
2019-09-05T17:56:18,793 [MQTT Call: things] INFO  o.e.k.i.w.l.Logger -   Record content:
2019-09-05T17:56:18,793 [MQTT Call: things] INFO  o.e.k.i.w.l.Logger -     mode : Manual
2019-09-05T17:56:18,794 [MQTT Call: things] INFO  o.e.k.i.w.l.Logger -     temperatureReported : 14
2019-09-05T17:56:18,794 [MQTT Call: things] INFO  o.e.k.i.w.l.Logger -     dateReported : Thu Sep 05 17:56:18 CEST 2019
2019-09-05T17:56:18,794 [MQTT Call: things] INFO  o.e.k.i.w.l.Logger -     humidityReported : 14
2019-09-05T17:56:18,795 [MQTT Call: things] INFO  o.e.k.i.w.l.Logger -
2019-09-05T17:57:18,735 [] INFO  o.e.k.c.d.DataServiceImpl - Storing message on topic: #account-name/#client-id/kura-gateway/shadow/update, priority: 7
2019-09-05T17:57:18,742 [] INFO  o.e.k.c.d.DataServiceImpl - Stored message on topic: #account-name/#client-id/kura-gateway/shadow/update, priority: 7
2019-09-05T17:57:18,743 [] INFO  o.e.k.d.h.Heater - Published message: org.eclipse.kura.message.KuraPayload@43e677
2019-09-05T17:57:18,746 [DataServiceImpl:Submit] INFO  o.e.k.c.d.t.m.MqttDataTransport - Publishing message on topic: aws/things/kura-gateway/shadow/update with QoS: 0
2019-09-05T17:57:18,811 [MQTT Call: things] INFO  o.e.k.c.c.CloudServiceImpl - Message arrived on topic: aws/things/kura-gateway/shadow/update
2019-09-05T17:57:18,814 [MQTT Call: things] INFO  o.e.k.i.w.l.Logger - Received WireEnvelope from org.eclipse.kura.wire.CloudSubscriber-1567686495335-4
2019-09-05T17:57:18,815 [MQTT Call: things] INFO  o.e.k.i.w.l.Logger - Record List content:
2019-09-05T17:57:18,815 [MQTT Call: things] INFO  o.e.k.i.w.l.Logger -   Record content:
2019-09-05T17:57:18,815 [MQTT Call: things] INFO  o.e.k.i.w.l.Logger -     mode : Manual
2019-09-05T17:57:18,816 [MQTT Call: things] INFO  o.e.k.i.w.l.Logger -     temperatureReported : 14.25
2019-09-05T17:57:18,816 [MQTT Call: things] INFO  o.e.k.i.w.l.Logger -     dateReported : Thu Sep 05 17:57:18 CEST 2019
2019-09-05T17:57:18,817 [MQTT Call: things] INFO  o.e.k.i.w.l.Logger -     humidityReported : 14.25
2019-09-05T17:57:18,817 [MQTT Call: things] INFO  o.e.k.i.w.l.Logger -
pi@raspberrypi:~ $
```

Figura 20: Fichero /var/log/kura.log

4.3 Bundle implementado en Eclipse Kura: Heater

Como ya se comentó en la sección 3.1.1.1 Bundles, los bundles son unos JAR y un manifiesto. Para implementar estos bundles, es necesario utilizar Eclipse como entorno de desarrollo [26]. Para desarrollar nuestros bundles se han utilizado los ejemplos que aparecen en esta documentación.

La clase que se encarga de obtener la información de los sensores y de implementar los métodos necesarios para el funcionamiento de cada uno de los servicios que utiliza y exporta es la clase `Heater.java`.

A continuación, se van a detallar los ficheros de metadatos del bundle, los servicios que esta usa y su funcionamiento.

4.3.1 Ficheros de metadatos

Proporcionarán información acerca de nuestro bundle. Estos ficheros son: MANIFEST.MF y component.xml

4.3.1.1 MANIFEST.MF

El fichero MANIFEST.MF de nuestro proyecto es el siguiente:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: org.eclipse.kura.demo.heater
Bundle-SymbolicName: org.eclipse.kura.demo.heater;singleton:=true
Bundle-Version: 1.0.600.qualifier
Bundle-Vendor: Eclipse Kura
Bundle-RequiredExecutionEnvironment: JavaSE-1.8
Service-Component: OSGI-INF/*.xml
Bundle-ActivationPolicy: lazy
Import-Package: org.eclipse.kura.cloud;version="[1.0,2.0)",
 org.eclipse.kura.cloudconnection.listener;version="[1.0,2.0)",
 org.eclipse.kura.cloudconnection.message;version="[1.0,2.0)",
 org.eclipse.kura.cloudconnection.publisher;version="[1.0,2.0)",
 org.eclipse.kura.configuration;version="[1.0,2.0)",
 org.eclipse.kura.message;version="[1.0,2.0)",
 org.osgi.service.component;version="1.2.0",
 org.slf4j;version="1.6.4"
```

Con la cabecera **`Manifest-Version`** indicamos la versión de la especificación para ficheros manifiesto que sigue el fichero. En este caso la 1.0.

El resto de las cabeceras que comienzan por **`Bundle-`** nos especifican características de OSGi. Las más destacadas son:

- **`Bundle-Name:`** Especifica el nombre del bundle.
- **`Bundle-SymbolicName:`** Especifica el nombre del paquete que identifica al bundle.
- **`Bundle-Version:`** Especifica la versión del bundle.
- **`Bundle-RequiredExecutionEnvironment:`** Especifica la versión de Java con la que se ejecuta.

Con la cabecera **`Service-Component`**: especificamos la ruta local al proyecto donde se sitúa un fichero de definición del componente `component.xml`.

Mediante la cabecera **`Import-Package`** indicamos las dependencias necesarias del workspace de Kura para el funcionamiento de nuestro Bundle.

4.3.1.2 component.xml

Este fichero nos permite indicar cual es la clase principal del bundle, los servicios que implementa, exporta y utiliza. En nuestro proyecto es el siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.2.0"
    name="org.eclipse.kura.demo.heater.Heater"
    activate="activate"
    deactivate="deactivate"
    modified="updated"
    enabled="true"
    immediate="true"
    configuration-policy="require">
    <implementation class="org.eclipse.kura.demo.heater.Heater"/>

   <property name="service.pid" type="String"
value="org.eclipse.kura.demo.heater.Heater"/>
    <service>
        <provide
interface="org.eclipse.kura.configuration.ConfigurableComponent"/>
    </service>

    <reference name="CloudPublisher"
            policy="dynamic"
            bind="setCloudPublisher"
            unbind="unsetCloudPublisher"
            cardinality="0..1"

interface="org.eclipse.kura.cloudconnection.publisher.CloudPublisher"/
>
</scr:component>
```

El elemento raiz es `scr:component`. Mediante el elemento `implementation` indicamos que clase es nuestra clase componente. Esta clase que se encarga de implementar los métodos `activate` y `deactivate` para arrancar y parar el bundle.

Este además contiene un elemento `service` por cada servicio que implementa y exporta. Contiene un elemento `reference` por cada servicio que utiliza. En este caso implementa y exporta `ConfigurableComponent` y utiliza el servicio `CloudPublisher`.

En cada elemento `service`, indicamos mediante el elemento `provide` cuál es la interfaz del servicio que se implementa.
En nuestro caso: `org.eclipse.kura.configuration.ConfigurableComponent.`

En cada elemento reference, indicamos mediante el atributo:

- `name`: El nombre del servicio.
- `policy`: Política asociada.
- `bind`: Método llamado para obtener la referencia del servicio.
- `unbind`: Método llamado para eliminar la referencia del servicio.

Por lo tanto, la clase `Heater` debe:

- Implementar la interfaz `ConfigurableComponent`.
- Implementar los métodos `activate`, `deactivate` y `update`.
- Implementar los métodos `setCloudPublisher` y `unsetCloudPublisher`.

4.3.2 Servicios implementados: ConfigurableComponent

Este es el servicio que nuestro bundle implementa y exporta. La implementación de este bundle se ha hecho teniendo como modelo el tutorial que proporciona Eclipse Kura [27].

En Eclipse Kura, al implementar la interfaz `ConfigurableComponent`, damos la posibilidad a nuestro bundle de exponer su configuración a través del servicio de configuración `ConfigurationService`.

Este asume que para cada componente configurable existe un recurso `xml` con metainformación acerca de su configuración en la carpeta OSGI- INF/metatype/ del componente. Es decir, para nuestro bundle `Heater` es necesario el fichero `org.eclipse.kura.demo.heater.Heater.xml`.

Una vez que el componente `configurable` está activo en el dispositivo Kura[11], si se ha configurado todo correctamente, podremos ver un nuevo apartado correspondiente a nuestro bundle en la interfaz web de Kura, a través del cual podremos actualizar el valor de los parámetros que hayamos indicado en el fichero `xml`. Esta interfaz se muestra en la Figura 21: Bundle Heater.

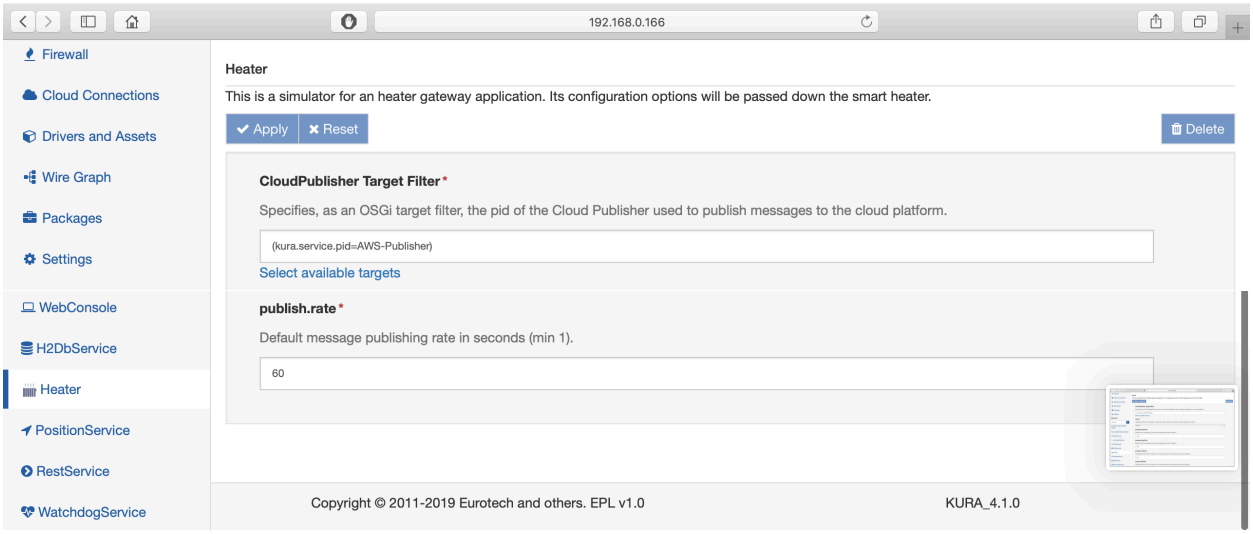


Figura 21: Bundle Heater

4.3.2.1 Fichero org.eclipse.kura.demo.heater.Heater.xml.

Este fichero xml de metainformación es el que necesita el servicio `ConfigurationService` para poder actualizar la interfaz Web y poder modificar los elementos del bundle implementado [27].

Este fichero debe contener un elemento `MetaData`, el cual debe contener un elemento `Desingate` y elementos opcionales `OCD` y `AD`. Los elementos `Designate` indican una asociación entre un elemento `OCD` y un pid (referencia a una clase) Los elementos `OCD` (Object Class Definition) contienen información sobre un conjunto de atributos y elementos `AD`. Los elementos `AD` (Attribute Definition) describen un atributo, la información de estos se mapea en cada uno de los elementos configurables.

A continuación, se muestra el contenido del fichero:

[11] Dispositivo en el que se ha instalado Eclipse Kura. En nuestro caso, las Raspberry Pi

```
<?xml version="1.0" encoding="UTF-8"?>
<MetaData xmlns="http://www.osgi.org/xmlns/metatype/v1.2.0"
localization="en_us">
     <OCD id="org.eclipse.kura.demo.heater.Heater"
          name="Heater"
          description="This is a simulator for an heater gateway
application. Its configuration options will be passed down the smart
heater. ">

          <Icon resource="OSGI-INF/heater.png" size="32"/>

          <AD id="CloudPublisher.target"
              name="CloudPublisher Target Filter"
              type="String"
              cardinality="0"
              required="true"
              default="(kura.service.pid=changeme)"
              description="Specifies, as an OSGi target filter, the pid
of the Cloud Publisher used to publish messages to the cloud
platform.">
          </AD>

          <AD id="publish.rate"
              name="publish.rate"
              type="Integer"
              cardinality="0"
              required="true"
              default="2"
              min="1"
              description="Default message publishing rate in seconds
(min 1)."/>
     </OCD>

     <Designate pid="org.eclipse.kura.demo.heater.Heater">
          <Object ocdref="org.eclipse.kura.demo.heater.Heater"/>
     </Designate>
</MetaData>
```

Como vemos, cada elemento `AD` tiene un `id`, este será el identificador que luego tendremos que utilizar en la clase Java `Heater` para asignar su valor a una variable. A continuación, se muestran las líneas de código que proporcionan esa asignación en la clase Java:

```
private static final String PUBLISH_RATE_PROP_NAME = "publish.rate";
private CloudPublisher cloudPublisher;
```

Como podemos ver en la Figura 21: Bundle Heater, el contenido del fichero visto en este apartado es el que se ha utilizado para mostrar la información en la interfaz Web, gracias a la implementación de la interfaz `ConfigurableComponent`.

4.3.2.2 Configuración del bundle

La configuración del bundle la podemos hacer mediante la interfaz Web que proporciona Eclipse Kura.

Con el campo **CloudPublisher Target Filter** que aparece en la Figura 21: Bundle Heater, podemos elegir cual es el servicio que queremos utilizar para que los mensajes se publiquen. Posteriormente, en el fichero **org.eclipse.kura.demo.heater.Heater.xml** se especifica un identificador para este, que será el que utilizaremos para mapear la información en la clase Java.

Con el campo **publish.rate** indicaremos la frecuencia con la que se publicarán los mensajes. Actualmente está establecido a una medida por cada minuto.

4.3.3 Servicios implementados: CloudPublisher

Este servicio es el que utilizaremos para conectarnos a la nube y publicar información en un topic. Haremos uso de la información mostrada en el apartado 5.1.1 Interacción con el Objeto: Thing Shadow para elegir el topic en el que se va a publicar. En este apartado de mostrará cómo se ha implementado en el bundle Heater.

Como se indica en la Documentación de Eclipse Kura [28], para implementar un CloudPublisher se deben implementar los métodos:

```
public void setCloudPublisher(CloudPublisher cloudPublisher) {
    this.cloudPublisher = cloudPublisher;
    }

public void unsetCloudPublisher(CloudPublisher cloudPublisher) {
    this.cloudPublisher = null;
    }
```

Haciendo uso de este CloudPublisher, publicaremos la información relativa de la humedad y temperatura obtenida a través de los sensores DHT11. Además, se indicará la fecha en la que se ha obtenido. Para ello, se hará uso del siguiente código:

```
KuraPayload payload = new KuraPayload();
KuraMessage message = new KuraMessage(payload);

payload.addMetric("dateReported", new Date().toString());
payload.addMetric("temperatureReported", this.temperature);
payload.addMetric("humidityReported", this.humidity);

// Publish the message
   try {
       this.cloudPublisher.publish(message);
       logger.info("Published message: {}", payload);
   } catch (Exception e) {
       logger.error("Cannot publish message: {}", message, e);
   }
```

Lo que se hace es crear un objeto `KuraPayload` con el contenido que queremos enviar por el topic. Se crea un objeto de tipo `KuraMessage`, que tiene como contenido el payload creado anteriormente. Para publicarlo, se hace uso del método **publish(KuraMessage)** del cloudPublisher.

4.3.4 Obtención de la información: Clase Heater.java

Para obtener la información que los sensores almacenan en el fichero **temperatura.log**, es necesario hacer uso de las librerías `java.io.File` y `java.util.Scanner`.

Para la lectura, se hará uso del siguiente código:

```
File fichero = new File("/home/pi/fichero_temperatura.txt");
Scanner s = null;
String linea;
try {
    s = new Scanner(fichero);
    //Leemos linea a linea el fichero
    while (s.hasNextLine()) {
         this.linea = s.nextLine()//Guardamos la linea en un String
    }
} catch (Exception ex) {
    logger.info("Mensaje: " + ex.getMessage());
} finally {
    // Cerramos el fichero tanto si la lectura ha sido correcta o no
    try {
        if (s != null)
            s.close();
    } catch (Exception ex2) {
        logger.info("Mensaje 2: " + ex2.getMessage());
    }
}
```

Posteriormente, se les asigna esta información a las variables `temperatura` y `humedad`. Para ello, recorremos el `String linea` y asignamos la información contenida a las variables.

4.4 Kura Wires y GPIODriver: Control de LEDs

Para controlar nuestro terrario, utilizaremos componentes que se suministran como parte del paquete de recursos Eclipse Kura. Uno de ellos es Kura Wires y el otro GPIODriver [29].

Lo primero que debemos hacer es configurar e instanciar GPIODriver para que podamos interactuar con los pines GPIO de la RaspberryPi. En estos pines, conectaremos un LED tal y como se indica en el Anexo A: Instalación de Sensores DHT11 y Leds en Raspberry Pi. Estos LEDs representarán posibles distintos elementos del terrario, como pueden ser un radiador o una luz. Sobre estos, actuaremos posteriormente desde la aplicación Web permitiendo encenderlos o apagarlos.

Para ello debemos seguir los pasos:

1. Instalar GPIODriver desde el Eclipse Marketplace.

Debemos copiar la URL proporcionada en el Eclipse Marketplace en el campo URL que se muestra en la Figura 22: Instalar GPIODriver. Para llegar a este, desde la interfaz Web de Eclipse Kura debemos ir a Packages -> Install/Upgrade.

```
http://download.eclipse.org/kura/releases/3.2.0/org.eclipse.kura.driver.gpio-1.0.0.dp
```

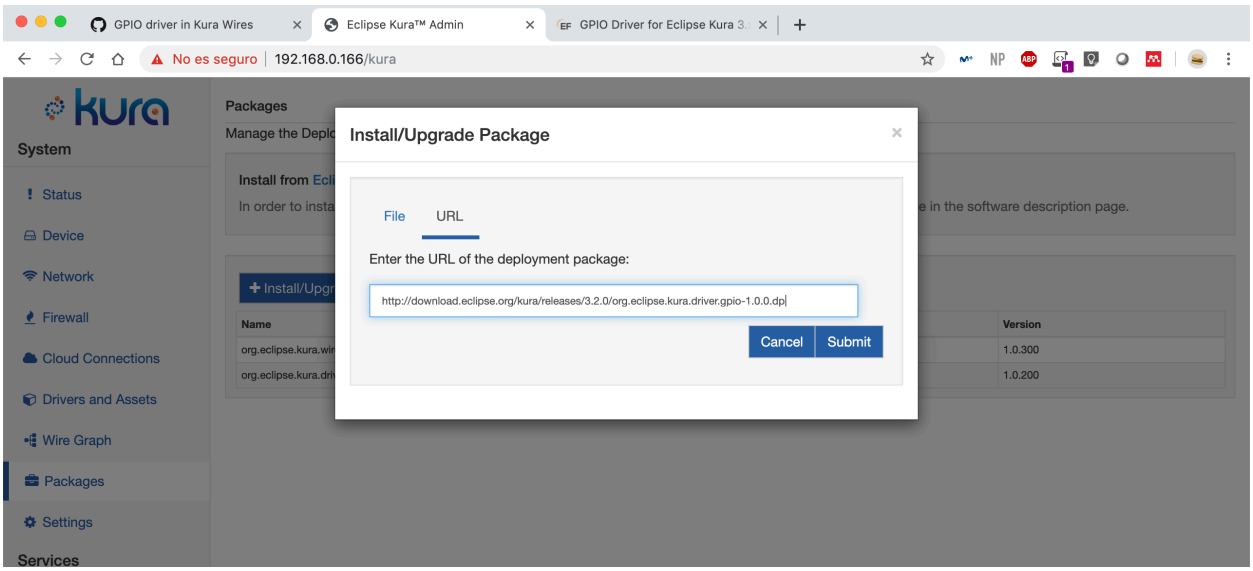


Figura 22: Instalar GPIODriver

2. Instanciar un GPIODriver

Para ello debemos ir desde la interfaz Web de Eclipse Kura a Drivers and Assets -> New Driver. Debemos seleccionar la información tal y como se muestra en la Figura 23: Instanciar GPIODriver.

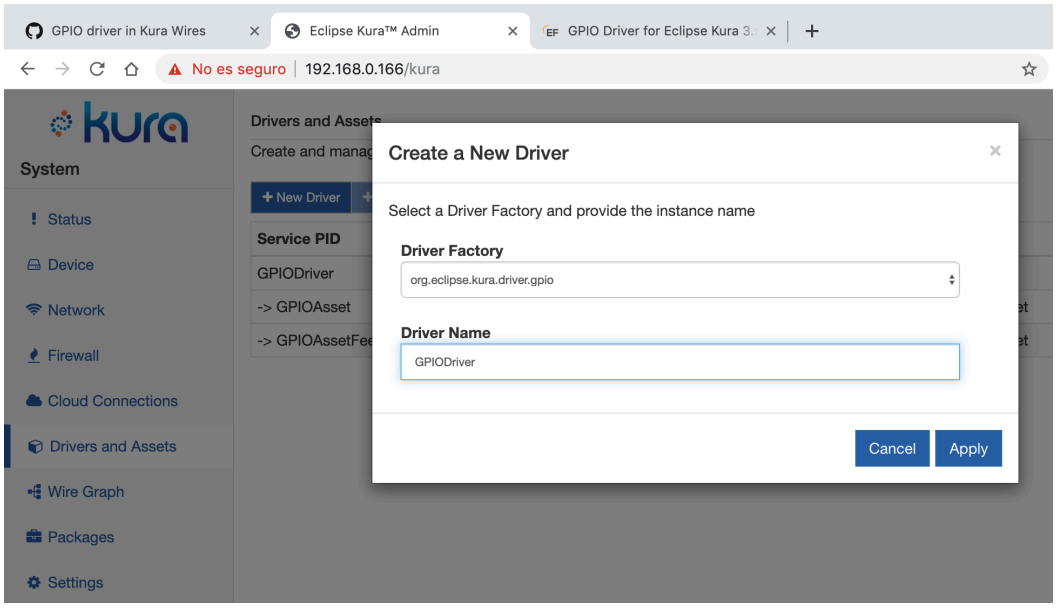


Figura 23: Instanciar GPIODriver

3. Crear GPIOAssets para interactuar con los LEDs.

Por último, en el mismo recurso de la interfaz Web que nos encontramos, hacemos click en el GPIODriver que acabamos de instanciar y posteriormente en New Asset y le ponemos el nombre que queramos.

Se nos debe abrir un desplegable como el que se muestra en la Figura 24: GPIO L. Lo configuraremos tal y como se muestra en esta. En **type** lo marcaremos como WRITE, ya que queremos escribir en el pin un **boolean** que indique si hay que encender o apagar el LED. En **resource.name** indicaremos el pin GPIO al que tenemos conectado el LED. Tal y como se indica en el Anexo A: Instalación de Sensores DHT11 y Leds en Raspberry Pi, es el GPIO 21. El campo **resource.direction** lo indicamos como OUTPUT, ya que lo que queremos hacer es escribir en este campo.

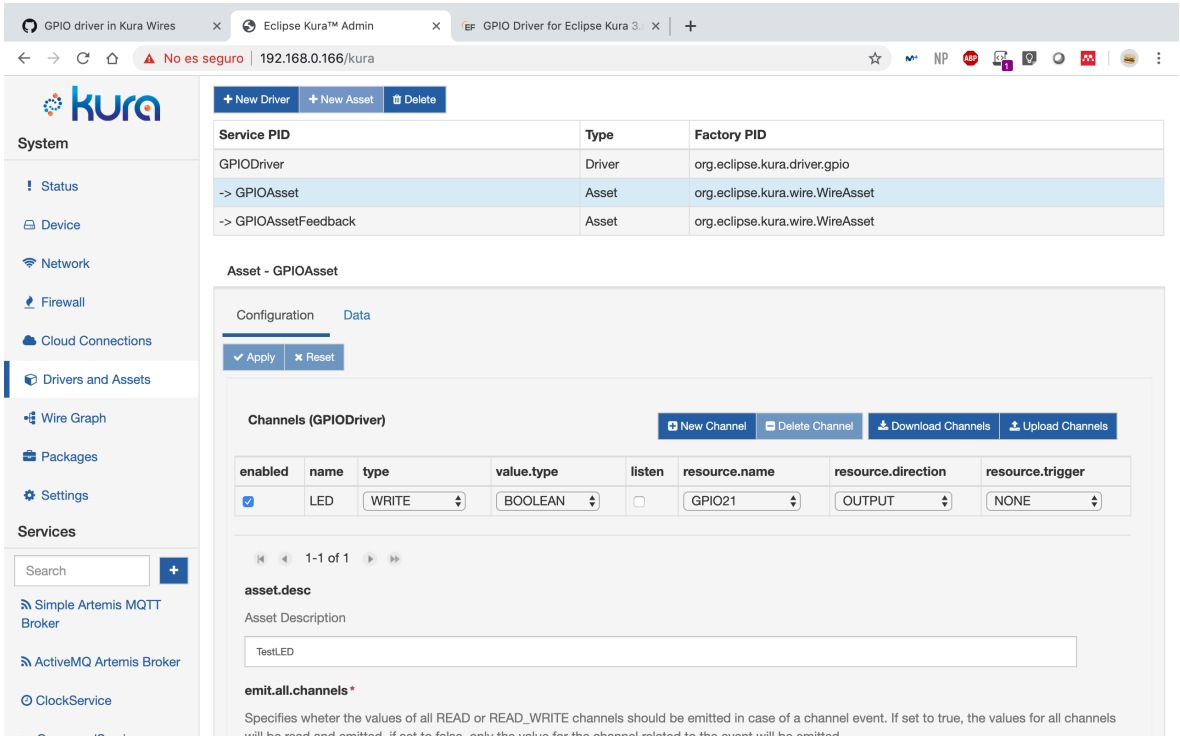


Figura 24: GPIO LED

Posteriormente, debemos irnos al apartado Wire Graph de la interfaz Web de Eclipse Kura. Ahí estableceremos las conexiones entre la información recibida del CloudSuscriber creado en el apartado 4.2.2 CloudSuscriber y el GPIOAsset del LED.

Para que el LED se encienda o apague dependiendo de la información obtenida en la suscripción, será necesario un script que sirva como controlador. Para ello en Wire Components seleccionaremos Javascript filter. Lo llamaremos Controller, tal y como se muestra en la Figura 25: Kura Wires.

Como se puede ver en esta, tendrá como entrada la información del CloudSuscriber y tendrá como salida una conexión con el GPIOAsset que hemos configurado anteriormente. Además, se conectará con el servicio **Logger** para escribir en el fichero **/var/log/kura.log** toda la información recibida y enviada.

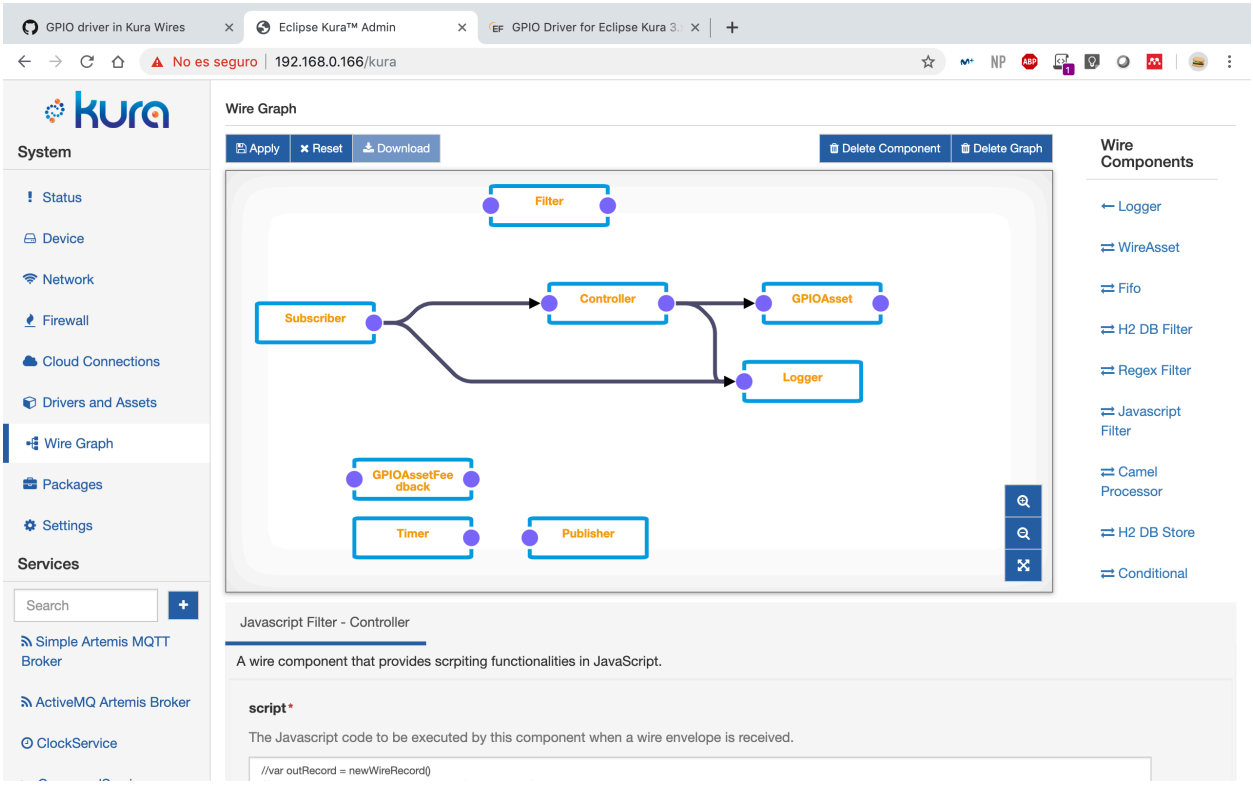


Figura 25: Kura Wires

El contenido del script es el siguiente:

```
  var record = input.records[0] //Se obtiene la información del
Suscriber

  for (var prop in record) { // Se itera en el contenido recibido

  if(record[prop].getType()==BOOLEAN){ //Se busca el campo con
información booleana
      logger.info('{}: {}', prop, record[prop])

      var counterRecord = newWireRecord() //Flujo de salida

      counterRecord.LED = record[prop] // Se asigna el valor al Led

      output.add(counterRecord) //Se inserta el valor en el flujo de
salida

      }
  }
```

De esta forma, se consigue encender/apagar el LED conectado al pin GPIO 21 cuando se reciba información de control (boolean) en el topic al que está suscrito el CloudSuscriber. En las figuras: Figura 26: Control enciende LED, Figura 27: Control apaga LED, se refleja el mensaje MQTT que produce este cambio.

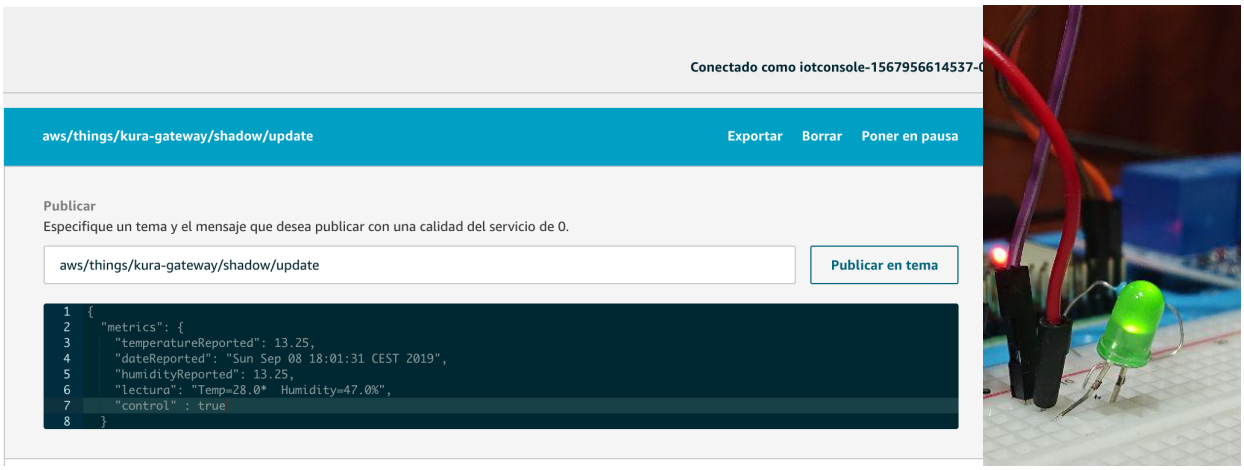


Figura 26: Control enciende LED

Figura 27: Control apaga LED

5 APLICACIÓN DESARROLLADA: CLOUD COMPUTING EN AMAZON AWS

If your business is not on the Internet, then your business will be out of business.

- Bill Gates -

Gran cantidad del desarrollo realizado en este trabajo está directamente implementado sobre los servicios que proporciona el entorno de Amazon Cloud Computing. En este capítulo se analiza cómo se ha conseguido desarrollar en este entorno concreto tanto el conexionado de los servicios que ofrece, como el código que lo hace posible. En el Anexo C: Creación de usuarios y configuración de servicios en AWS**,** se muestra la información necesaria para registrarnos en AWS, así como los pasos necesarios para crear cada uno de los servicios utilizados.

5.1 IoT Core: Puerta de acceso al Cloud Computing para un Gateway IoT.

Nuestro sistema está pensado para que pueda ser usado tanto por un único terrario, como por múltiples terrarios que podrían llegar a agruparse en tiendas. Para simplicidad y facilidad, se va a suponer que un terrario contiene una RaspberryPi, junto con los sensores necesarios para obtener la información. Así es como se muestra en la Figura 1: Esquema de la arquitectura.

Es por esto por lo que es de vital importancia separar la información de cada terrario. Del mismo modo, es necesario que cada terrario tenga una identidad y que esto le permita manejar su información de manera segura, es decir, que nadie que no tenga permisos para acceder a la información de este pueda enviar u obtener información relativa a él.

Para ello se hace uso del componente **Registro** que proporciona IoT Core.

Cada terrario se creará en el sistema como un "Objeto" en el registro de "Objetos". Para más información de cómo registrar un objeto en IoT Core, consultar el apartado 7.4 IoT Core del Anexo C: Creación de usuarios y configuración de servicios en AWS.

Como podemos observar, hemos creado un Objeto denominado "**kura-gateway**" correspondiente a una Raspberry Pi con Eclipse Kura instalado. Para cada objeto, en su creación se crea y se activa un certificado, del cual podemos obtener su clave privada. Esto se muestra en la Figura 28: Certificado asociado a un objeto.

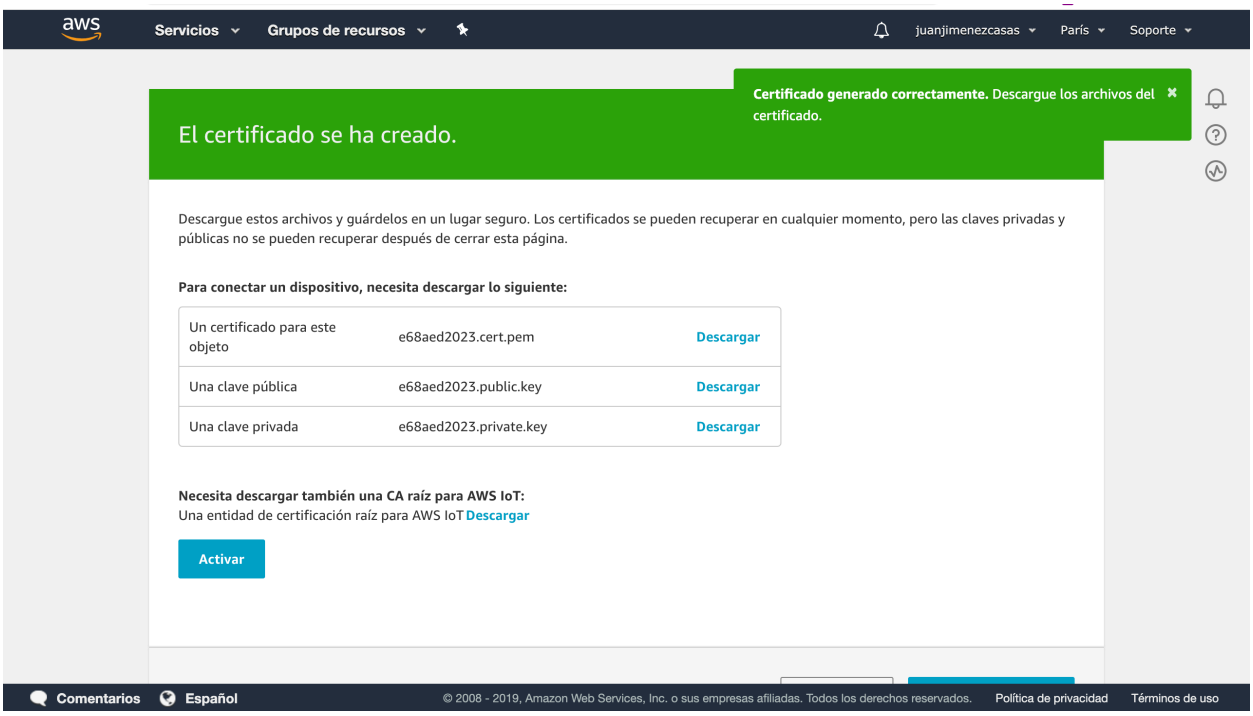


Figura 28: Certificado asociado a un objeto

Con este certificado, el Objeto adquiere una Identidad, que utiliza el componente **Servicio de seguridad e identidad** para que la Raspberry Pi pueda conectarse a la nube y acceder a la información en su nombre. La funcionalidad del servicio de seguridad e identidad es la que se detalla en el apartado 3.2.1 AWS IoT Core

A este certificado se le van a asociar unas Políticas personalizadas, que nos van a permitir restringir el número de Raspberry Pi que pueden publicar en un Objeto. **Para su simplicidad, se tendrá en cuenta la consideración realizada anteriormente y a cada objeto asignaremos una Raspberry Pi.**

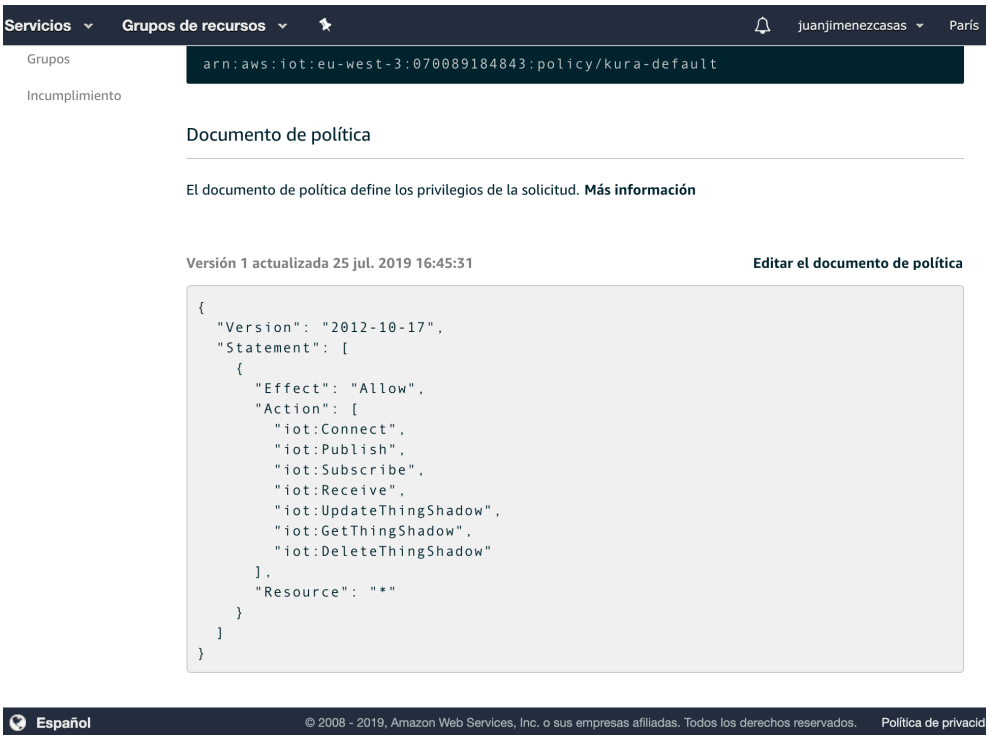


Figura 29: Política asociada a un certificado

Como vemos en la Figura 29: Política asociada a un certificado, el documento de una política se escribe en formato JSON, y su objeto **Statement** contiene una serie de reglas en las que se especifica si se permite o no (**Effect**) una operación (**Action**) sobre un recurso (**Resource**) [30].

Si quisiéramos asignar una restricción de los dispositivos que se pueden conectar bastaría con cambiar a " Resource" : " topic_publicación_eclipse_kura[12] "

5.1.1 Interacción con el Objeto: Thing Shadow

Para cada objeto se crea una **Sombra** de este, haciendo uso del componente **Sombra del Dispositivo**, que no es más que un documento JSON que nos indica el estado de un objeto.

[12] topic_publicacion_eclipse_kura: Ruta del topic MQTT en el que se quiera publicar/suscribir.

Estado de sombra:


```
{
  "desired": {
    "moisture": "okay",
    "dateDesired": 1566494048455,
    "temperatureDesired": "34",
    "humidityDesired": "23",
    "dateReported": 1566301545529,
    "id": 56,
    "name": "prueba"
  },
  "reported": {
    "moisture": "okay",
    "dateReported": "2131823192",
    "temperatureReported": "15",
    "humidityReported": "45"
  },
  "delta": {
    "dateDesired": 1566494048455,
    "temperatureDesired": "34",
    "humidityDesired": "23",
    "dateReported": 1566301545529,
    "id": 56,
    "name": "prueba"
  }
}
```


Figura 30: Sombra de un Objeto

Como podemos ver en la Figura 30: Sombra de un Objeto, cada Sombra puede tener varios estados, con información relativa a este.

- El estado **desired** corresponde a una información de estado deseado. En este campo es donde se modificará y almacenará la información procedente de la aplicación Web para controlar cada uno de los terrarios.
- El estado **reported** corresponde a una información de estado reportado. En este campo es donde se modificará y almacenará la información procedente del Dispositivo Kura, ya que contendrá la información reportada por cada uno de los sensores.
- El estado **delta** corresponde a una información de estado variante. En este campo se incluye la información que se ha modificado en la última actualización de una Sombra. Es de gran utilidad para comprobar que todo funciona según lo esperado y no se ha producido ningún error.

Para poder interactuar con los distintos objetos creados utilizaremos el componente **Servicio Device Shadow.** La funcionalidad de este se indica en el apartado 3.2.1 AWS IoT Core. Este componente nos proporciona la posibilidad de actualizar y obtener información de una Sombra mediante:

1. Un punto de enlace a una API REST [31]:

Esta API es la que utilizaremos para actualizar la información de la sombra desde la aplicación Web. Los métodos que proporciona son: GET, UPDATE y DELETE.

2. Un tema MQTT para actualizar, obtener o eliminar la información de estado de una Sombra [32]:

En los temas MQTT que podemos ver en la Figura 31: Servicio Device Shadow, nos suscribiremos y publicaremos la información obtenida procedente/destinada a la Raspberry Pi.

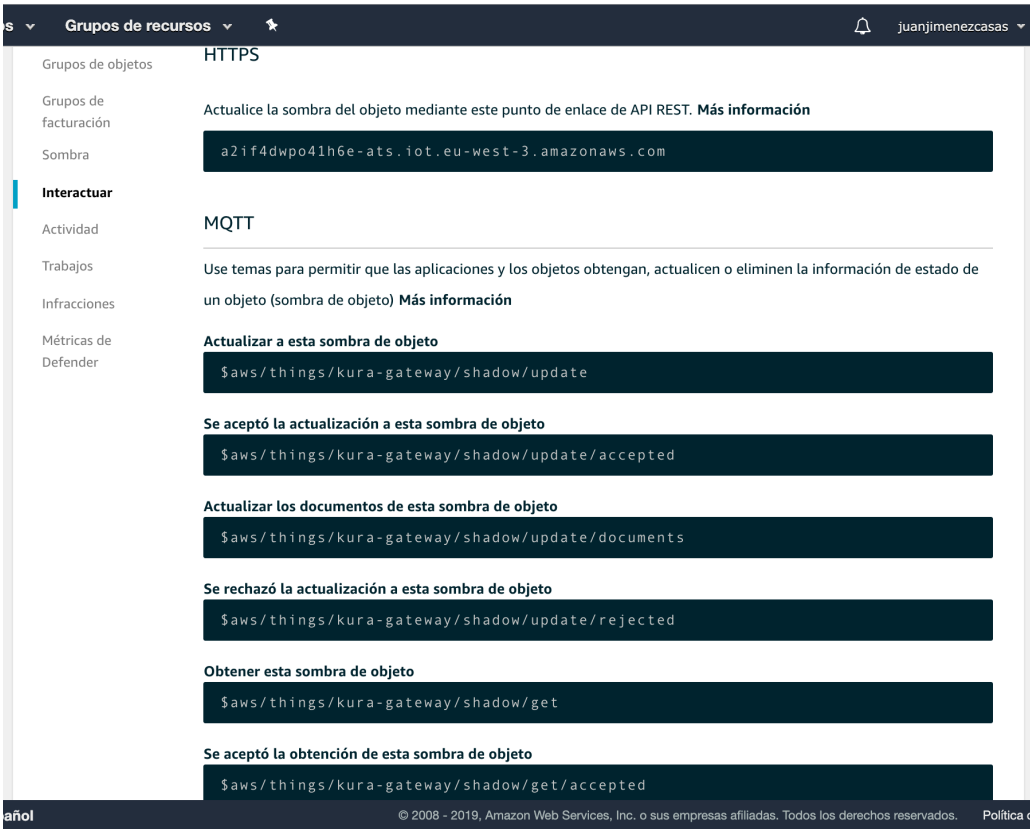


Figura 31: Servicio Device Shadow

5.1.2 Reglas

Las reglas son una parte fundamental para la interconexión entre los distintos servicios que proporciona AWS.

Estas son acciones que realizar cuando se produce un evento dado, ya sea un mensaje recibido/enviado por un objeto o actualizar la sombra de este. Estas reglas necesitan saber el origen de los mensajes, y los atributos que se quieren de este. Para facilitar su uso, estas consultas se realizan en lenguaje SQL:

```
SELECT <Attribute> FROM <Topic Filter> WHERE <Condition>
```

En la Figura 32: Acciones AWS, se muestran las diferentes acciones que se pueden hacer desde el IoT Core.

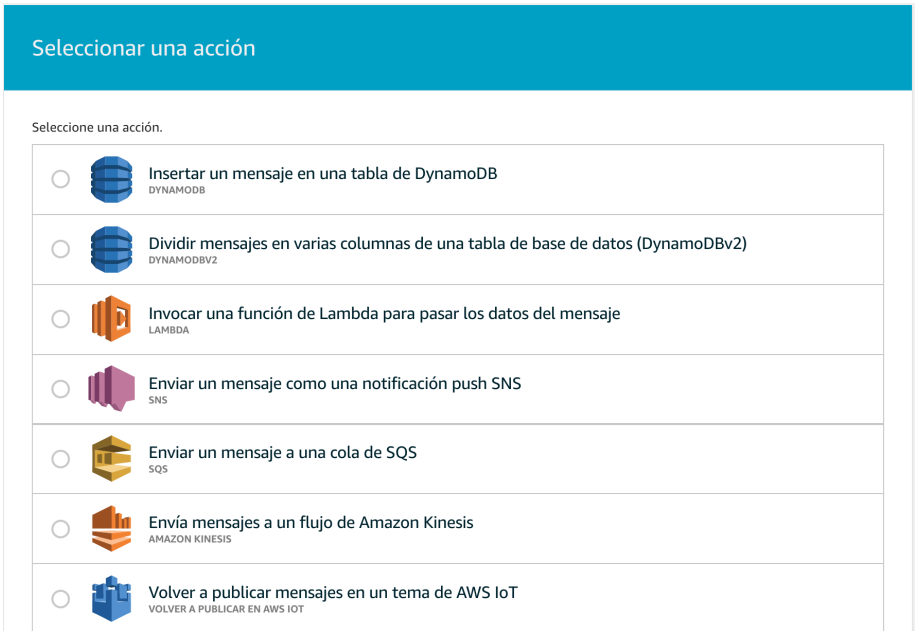


Figura 32: Acciones AWS

Para el desarrollo de nuestra plataforma, han sido de gran utilidad las reglas de **Insertar un mensaje en una tabla de DynamoDB**, así como **Invocar a una función Lambda para pasar los datos del mensaje**. El funcionamiento de estas se detallará en los apartados a continuación.

Para el desarrollo de nuestra aplicación, hemos creado una regla llamada **sendToDB**. Esta se encarga de invocar una función Lambda cuando se actualiza la Sombra de un Objeto.

Cuando se produce la regla, a la función Lambda se le pasa el contenido de esta sombra, así como el nombre del Objeto al cual se le está actualizando la Sombra. Esto se consigue con la sentencia SQL:

```
 SELECT state.reported.* , state.desired.* , topic(3) AS kuraId FROM
'$aws/things/+/shadow/update/accepted'
```

Ya que estaríamos haciendo un SELECT sobre el topic en el que se guarda la información actualizada de una Sombra ($aws/things/+/shadow/update/accepted) y extrayendo los parámetros de estado "deseado" y "reportado".

Como aclaración, el '+' es el elemento comodín para elegir el topic, (es como un *). Significa cualquiera. Esto es de gran utilidad de cara a la escalabilidad, ya que vamos a poder utilizar la misma regla para obtener la Sombra de todos los Objetos que tengamos creados. Para saber de qué objeto se ha obtenido la información, se obtiene también el parámetro topic (3), correspondiente al tercer elemento de la ruta del topic, que se correspondería con lo que se introduce en el elemento '+'.

En caso de que se produzca algún error se enviará un mensaje con los datos de error a CloudWatch[13].

Desde la interfaz Web de AWS, una vez se ha creado la regla, se nos debe mostrar la información contenida en la Figura 33: Regla SendToDB.

[13] CloudWatch es el servicio de eventos y alertas interno de Amazon AWS.

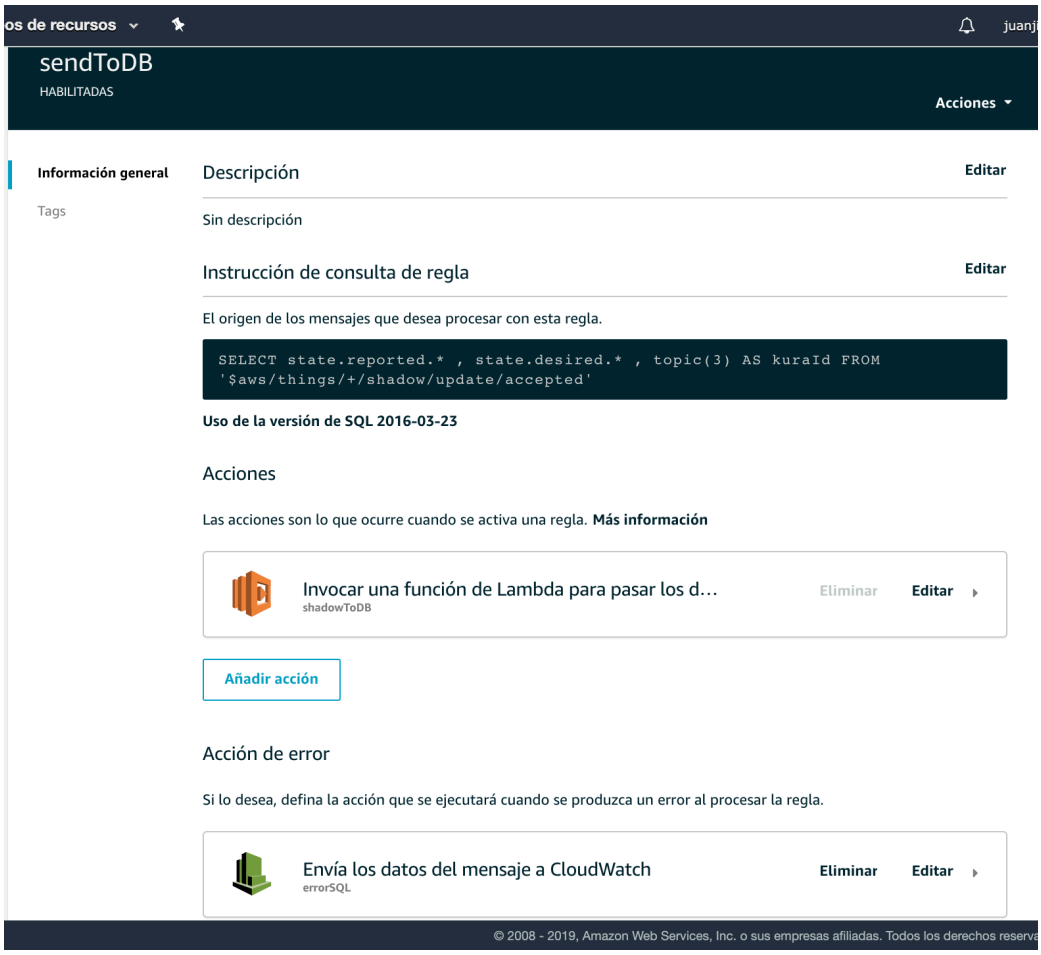


Figura 33: Regla SendToDB

5.2 Dynamo DB

Para almacenar la información perteneciente a los estados de las distintas Sombras, utilizaremos una Base de Datos noSQL. El objetivo de esta Base de Datos es la de almacenar un histórico de los distintos estados en los que se ha encontrado un terrario, para poder visualizarlo y actuar en consecuencia. Además, esta será la Base de Datos a la que se acceda mediante la API REST proporcionada por API Gateway.

Hemos creado una tabla llamada **terrario**. Esta tiene una clave primaria llamada **kuraId** (string) y una clave de ordenación llamada **date** (number). Se han elegido de esta forma ya que identificaremos a cada terrario por su nombre y utilizaremos la clave de ordenación para acceder al histórico de información por fechas de un terrario. Una vez hayamos creado la tabla, se nos debe mostrar la información que se muestra en la Figura 34: Información Base de Datos terrario.

La Base de Datos posee un ARN propio, que utilizaremos para asignar los distintos permisos necesarios para poder leer/escribir en esta desde el resto de los servicios de AWS. Tanto la creación de la Base de Datos como el ARN se ve en más detenimiento en el Anexo C: Creación de usuarios y configuración de servicios en AWS, apartado 7.5 DynamoDB.

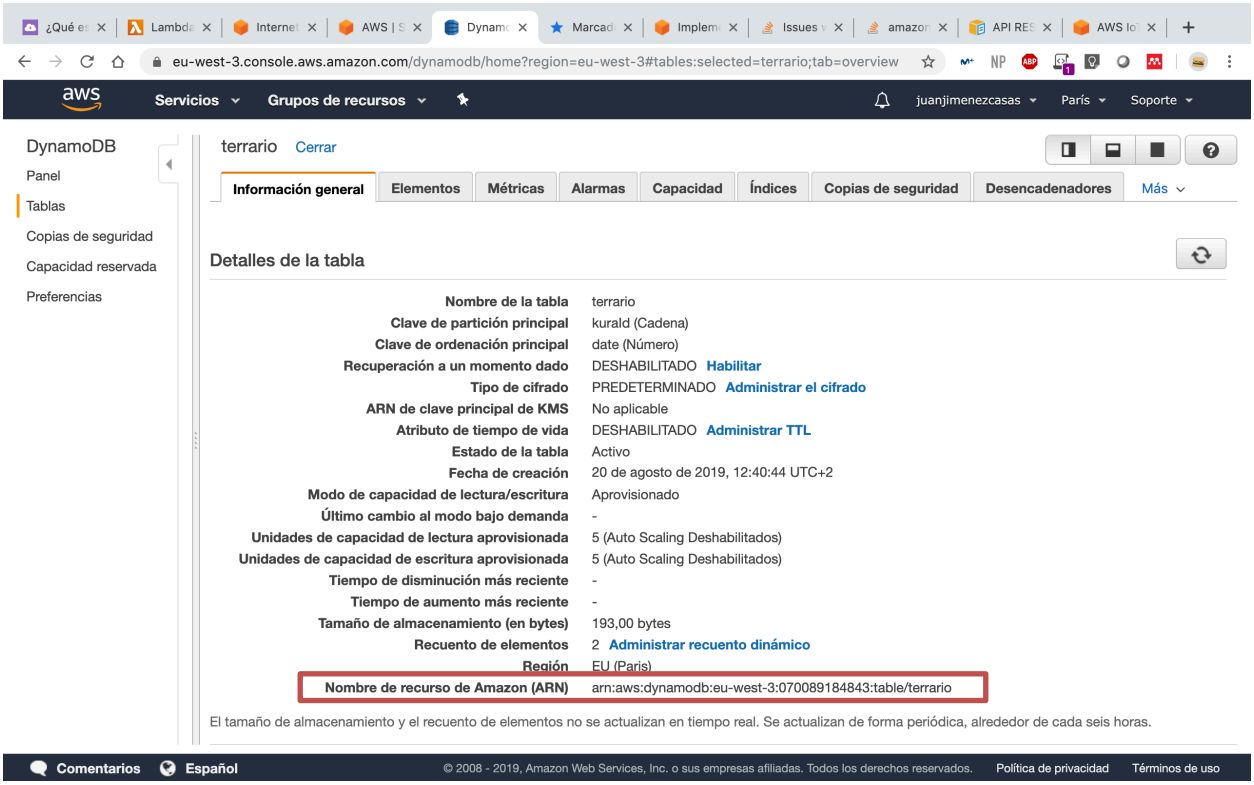


Figura 34: Información Base de Datos terrario

Para el uso que se ha definido, la tabla contendrá elementos con información:

- **kuraId (string):** Nombre identificativo que se le asigna al terrario.
- **date (number):** Fecha en la que se ha incluido la información.
- **name (string):** Nombre que indica información acerca del terrario, como el dueño o la tienda.
- **type (string):** Campo que indica si la información introducida es de tipo desired o reported.
- **humidityDesired (string):** Humedad deseada.
- **humidityReported (string):** Humedad reportada.
- **temperatureDesired (string):** Temperatura deseada.
- **temperatureReported (string):** Temperatura reportada.
- **moistureReported (string):** Campo que indica si ha habido alguna incidencia o no.
- **id (number):** Índice que hace referencia a un terrario en un momento dado. Es de utilidad para acceder a un estado dado del terrario desde la aplicación Web.

En la Figura 35: Elementos de terrario, podemos observar cómo se visualizan los distintos elementos que se incluyen en la tabla.

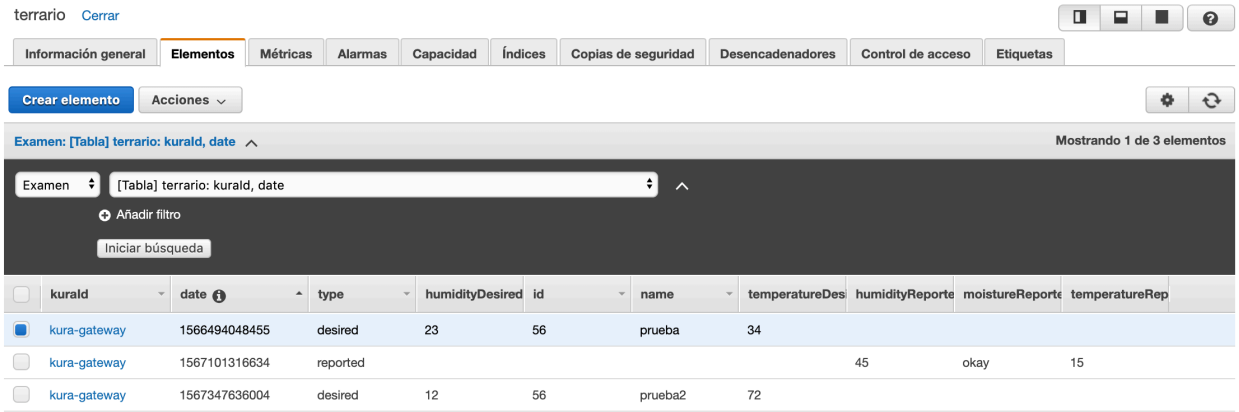


Figura 35: Elementos de terrario

5.3 Lambda Function

El punto clave de este trabajo se encuentra en este apartado. Gracias a las funciones Lambda que proporciona AWS, podemos procesar el contenido que recibimos de las sombras para almacenarlo en la Base de Datos u ofrecerlo en una API REST, para cualquier servicio, sin necesidad de crear un servidor. Esto se conoce en la actualidad como *aplicación serverless.*
Estas pueden ser escritas en varios lenguajes como Python o Node.js. Para conocer el funcionamiento de estas, vamos a basarnos en la siguiente ilustración Figura 36: Estructura de una función Lambda [33] de una función escrita en Node.js:

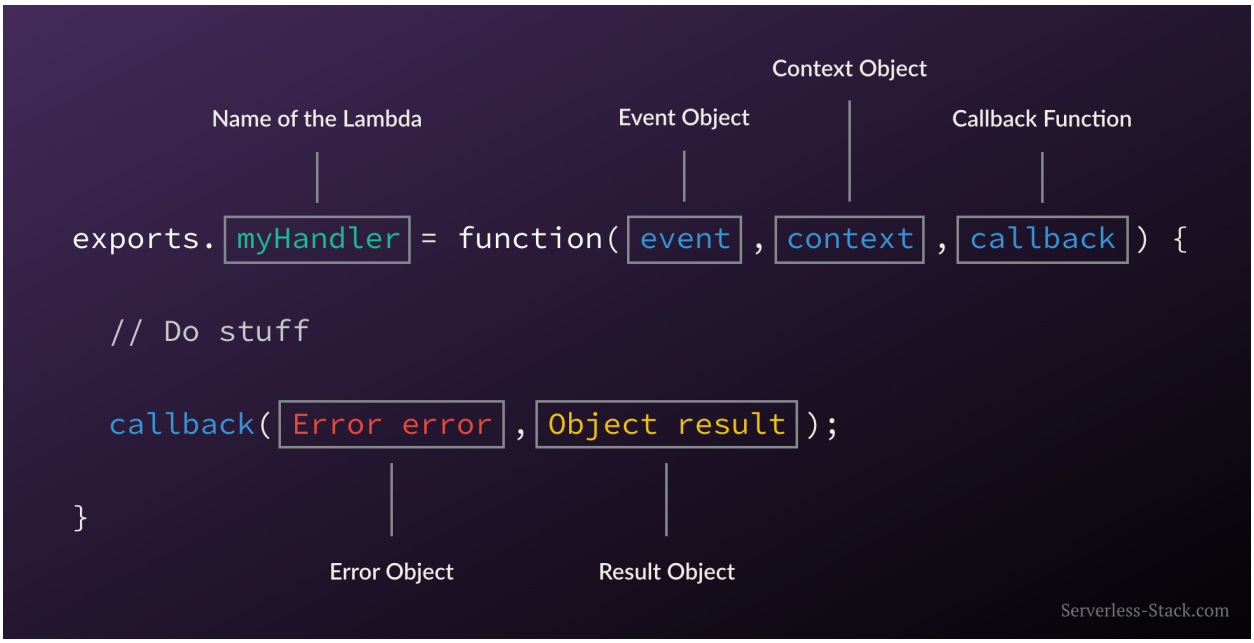


Figura 36: Estructura de una función Lambda

Como podemos ver, el nombre de nuestra función Lambda sería `myHandler`. El objeto `event` contiene toda la información sobre el evento que desencadenó esta función Lambda. En el caso de que sea una solicitud HTTP, contendrá la información específica sobre esta (cabeceras, contenidos en formato JSON). El objeto `context` contiene información sobre el entorno donde se está ejecutando nuestra función Lambda. Una vez la función ha finalizado, se llamará a la función `callback` con los resultados (o error) y AWS responderá a las solicitudes HTTP o eventos que la invoquen con él.

Sabiendo cómo funcionan este tipo de funciones, realizar un procesamiento de la información para posteriormente crear una API REST o transferir la información a la Base de Datos es tarea sencilla.

Para ello hemos creado tres funciones Lambda. Todas ellas tienen asociado el Rol de ejecución creado en el Anexo C: Creación de usuarios y configuración de servicios en AWS, apartado 7.6 Funciones Lambda. Este les permitirá tener los permisos necesarios para acceder y modificar los elementos de la Base de Datos DynamoDB, así como actualizar la sombra de un Objeto del IoT Core o introducir el contenido de esta en la Base de Datos cada vez que se actualice.

5.3.1 Función shadowToDB

Esta función es la que se encarga de almacenar la información de la Sombra de una Objeto en la Base de Datos.

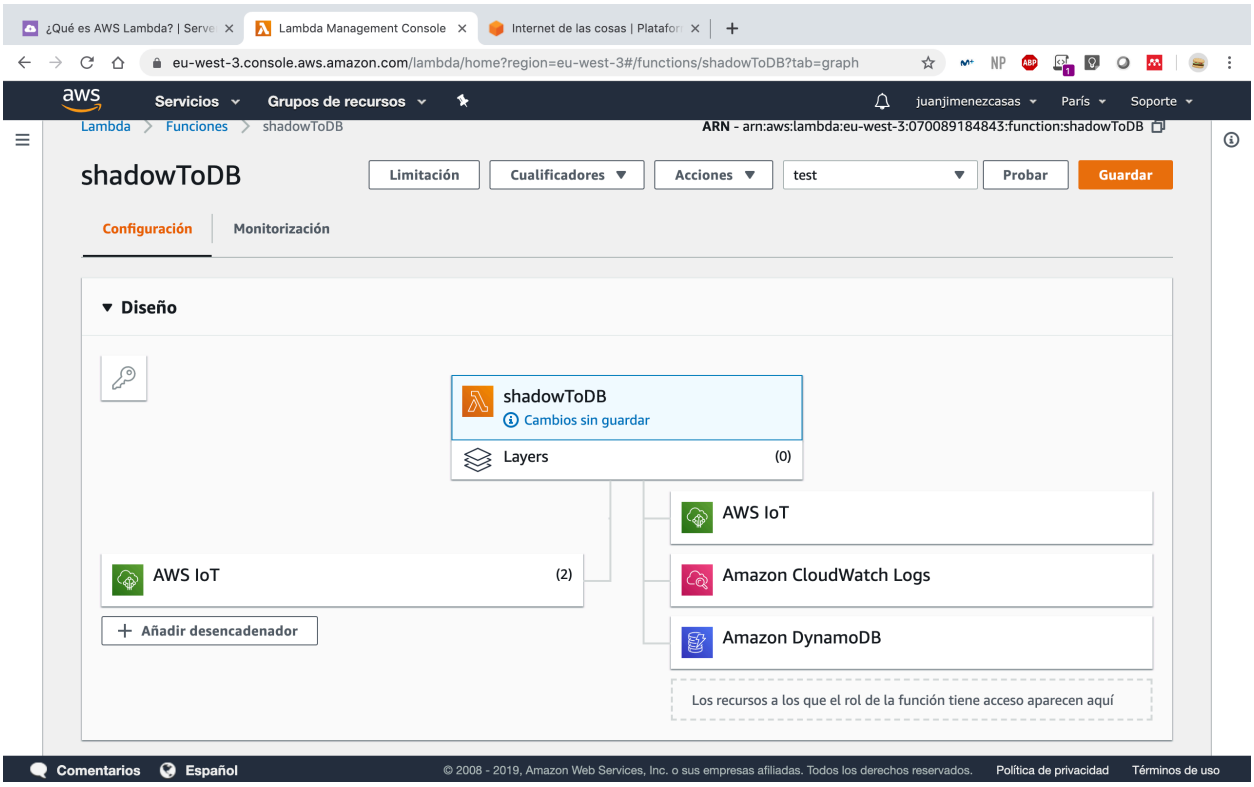


Figura 37: Función shadowToDB

Como podemos observar en la Figura 37: Función shadowToDB, tiene un evento desencadenador, la regla de IoT Core **sendToDB,** vista en el apartado anterior. Por lo tanto, el contenido del objeto `event` contendrá el estado de una Sombra, obtenido con la regla anteriormente descrita.

Como queremos obtener la información del **IoT Core** e introducir la información en la Base de Datos **DynamoDB**, así como introducir la información en los registros de **Amazon CloudWatch**, tendremos que darle permisos al rol de la función para que pueda acceder a estos. Esto se explica en el Anexo 3.

Para acceder a la Base de Datos, la función implementada necesita hacer uso de las líneas de código:

```
var dynamo = new AWS.DynamoDB.DocumentClient();
var table = "terrario";
```

Con la que `dynamo` hace referencia a la Base de Datos y con `table` indicamos el nombre de la tabla.

Para introducir la información, creamos una variable llamada `paramsPut`, con la que convertimos la información obtenida del objeto `event` en campos para la Base de Datos. El contenido de esta variable se muestra en la Figura 38: Variable paramsPut.

```
var paramsPut = {
 TableName:table,
 Item:{
     "kuraId": event.kuraId,
     "date": Date.now(),
     "type": "reported",
     "temperatureDesired" : event.temperatureDesired,
     "temperatureReported": event.temperatureReported,
     "humidityDesired" : event.humidityDesired,
     "humidityReported": event.humidityReported,
     "moistureReported" : event.moisture
     }
 };
```

Figura 38: Variable paramsPut

Para introducir la información en la Base de Datos, ejecutamos las siguientes líneas de código:

```
if(event.temperatureReported !=null &&event.humidityReported !=null){
        dynamo.put(paramsPut, function(err, data) {...});
}else {...}
```

En la que `{...}` corresponde a la creación de los eventos de registro con la información de éxito o fallo para la depuración del código.

Como resultado de esta función, cuando se actualiza la Sombra de un Objeto, se introduce una fila en la Base de Datos **DynamoDB**. En la Figura 39: Tabla Terrario se muestra como se ha añadido un campo en la tabla, tras una actualización.

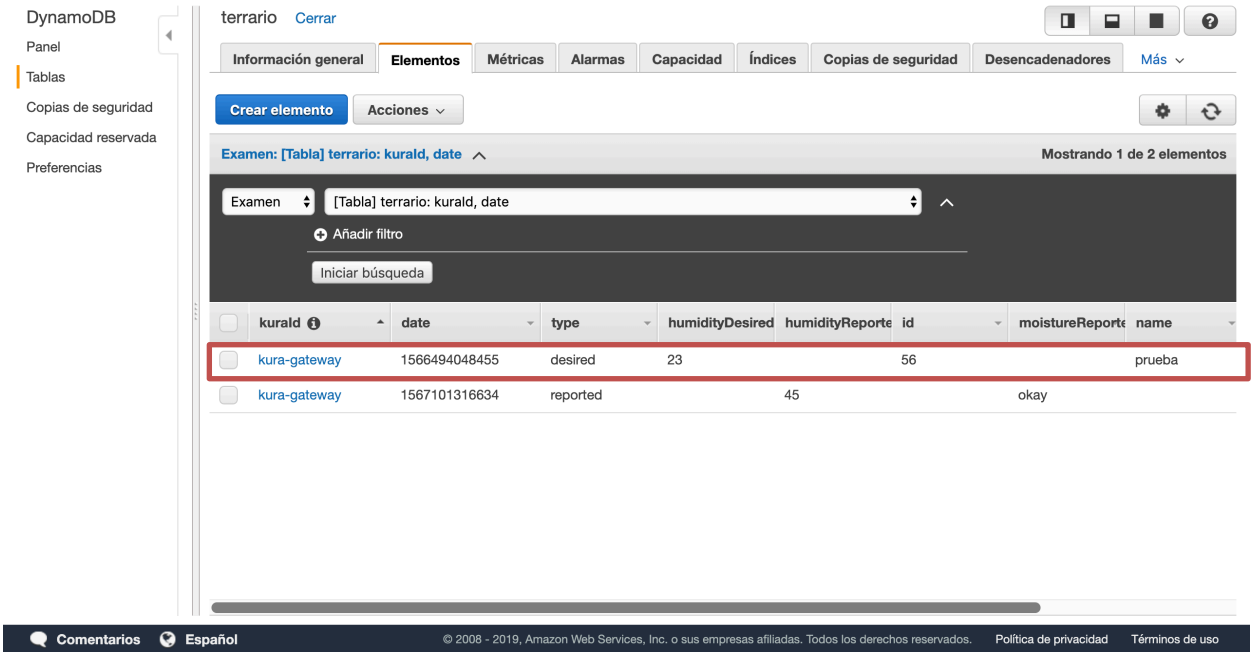


Figura 39: Tabla Terrario

5.3.2 Función dynamodb_read

Esta función es la que se encarga de obtener la información de la base de datos.

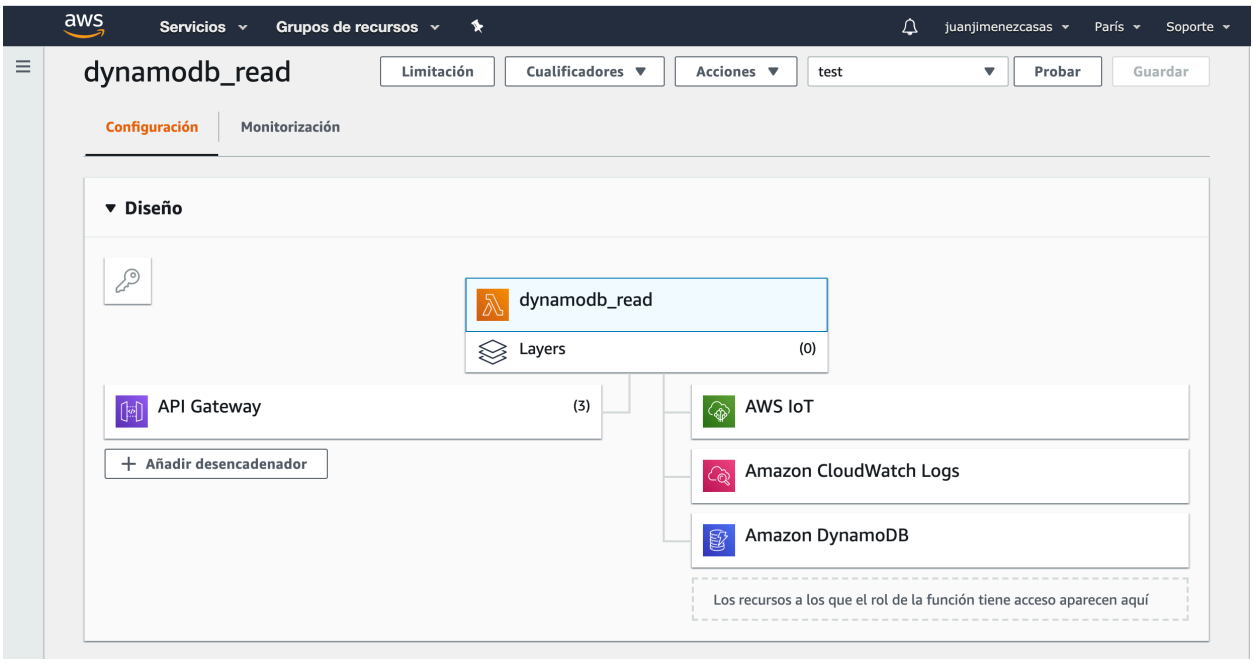


Figura 40: Función dynamodb_read

Como podemos observar en la Figura 40: Función dynamodb_read, tiene un evento desencadenador, la **API Gateway**. Por lo tanto, el contenido del objeto `event` contendrá la información relativa a la petición HTTP que se haga a esta. En el apartado de API Gateway se indica cómo se tiene que enviar esta información para que ambos extremos puedan funcionar. Se responderá a la petición HTTP con el resultado de la ejecución del código.

Para acceder a la Base de Datos lo hacemos de la misma forma que con la función anterior:

```
var dynamo = new AWS.DynamoDB.DocumentClient();
var table = "terrario";
```

Como en esta función lo que queremos es obtener la información de la Base de Datos, se proporcionan dos métodos para ello:

- **Método scan ():**

Nos permite obtener todos los elementos de una tabla. Es de gran utilidad cuando queremos obtener toda la información posible de la tabla, con el fin de obtener un histórico para la monitorización. Sólo tenemos que indicarle el nombre de la tabla y el número límite de elementos que queremos obtener. El contenido de este método se muestra en la Figura 41: Método scan.

```
9     let scanningParameters = {
10        TableName: 'terrario',
11        Limit: 100
12    };
13
14    docClient.scan(scanningParameters, function(err,data){
15        if(err){
16            callback(err,null);
17        }else{
18            callback(null,data);
19        }
20    });
```

Figura 41: Método scan

- **Método get ():**

Nos permite obtener los elementos de una tabla haciendo uso de la clave primaria de esta. De esta forma sólo obtenemos las filas necesarias. Es de gran utilidad cuando queremos obtener la información de un terrario dado. Tendremos que indicarle el nombre de la tabla en la que queremos consultar, así como la clave que nos servirá para encontrar la fila que queremos. El contenido de este método se muestra en la Figura 42: Método get.

```
22    var params = {
23        TableName: 'terrario',
24        Key: {
25            "kuraId": event.kuraId
26
27        }
28    };
29
30    //Esto busca solo por la clave que le indiquemos
31    docClient.get(params, function(err,data){
32       if(err){
33           callback(err,null);
34       }else{
35           callback(null,data);
36       }
37    });
```

Figura 42: Método get

Por lo tanto, si hacemos una petición GET a la URL que nos proporciona la API Gateway, que vemos en el apartado 5.4 API Gateway, obtendremos la información que contiene la Base de Datos. Para esta consulta hacemos uso de **Postman** y la función **scan ()** para obtener todo el contenido de la Base de Datos. El resultado de esta consulta se muestra en la
Figura 43: Consulta REST a la Base de Datos con Postman. Como podemos ver, coindice con el mostrado en la Figura 39: Tabla Terrario.

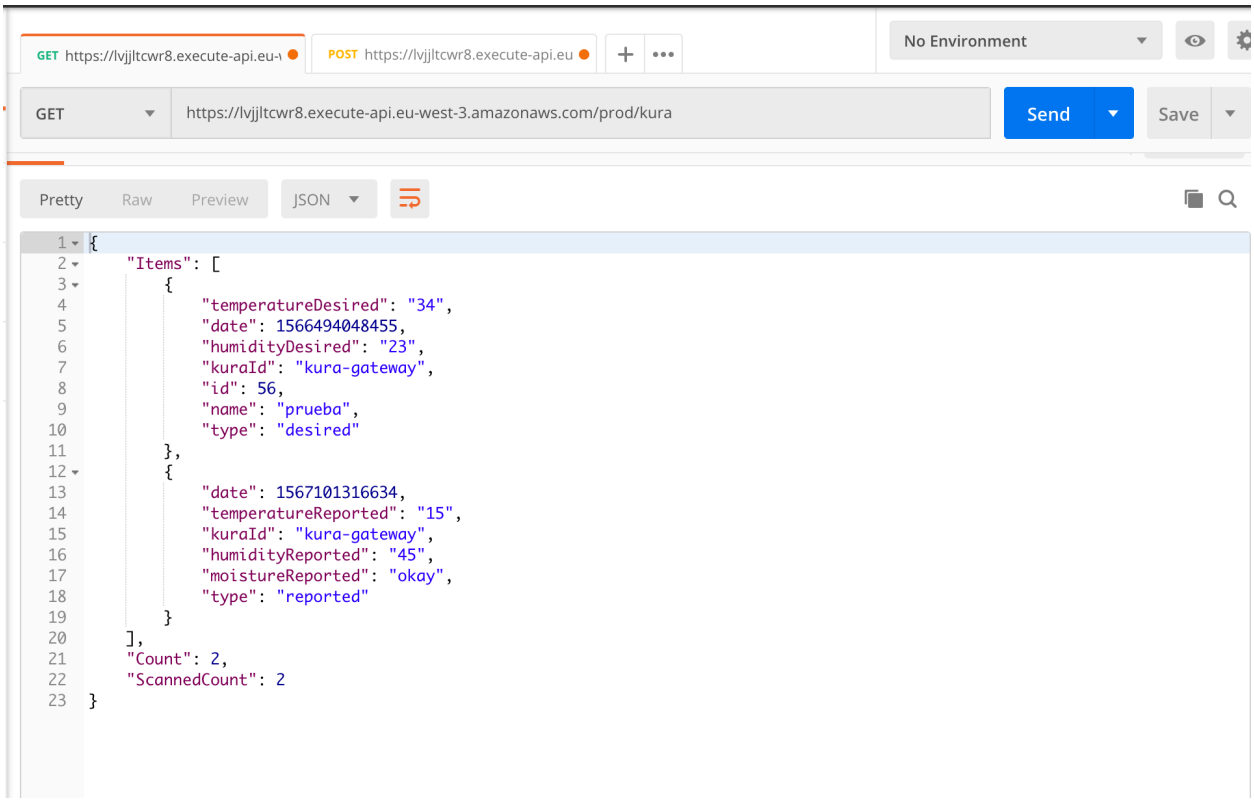


Figura 43: Consulta REST a la Base de Datos con Postman

5.3.3 Función dynamodb_write

Esta función es la que se encarga de actualizar el estado de una sombra, y, en consecuencia, introducir una fila en la Base de Datos con el nuevo estado.

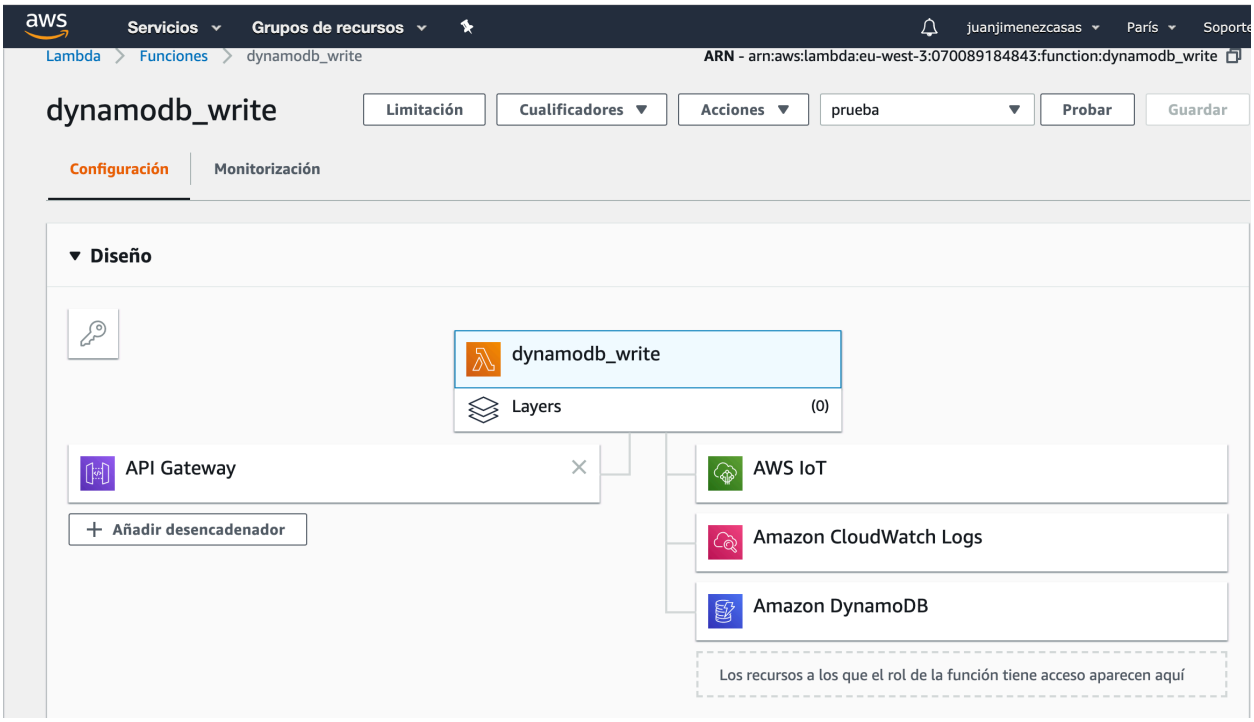


Figura 44: Función dynamodb_write

Como podemos observar en la Figura 44: Función dynamodb_write, tiene un evento desencadenador, la **API Gateway**. Por lo tanto, el contenido del objeto `event` contendrá la información relativa a la petición HTTP que se haga a esta. En el apartado de API Gateway se indica cómo se tiene que enviar esta información para que ambos extremos puedan funcionar. Se responderá a la petición HTTP con el resultado de la ejecución del código.

Como esta función se encarga de actualizar el estado de la Sombra, no es necesario que acceda a la Base de Datos, pero si a la Sombra del Objeto. Para ello, son necesarias las líneas de código:

```
const AWS = require('aws-sdk');

var config = {
    "thingName":"kura-gateway",
    "endpointAddress":"a2if4dwpo41h6e-ats.iot.eu-west3.amazonaws.com"
};

var iotdata = new AWS.IotData({endpoint: config.endpointAddress});
```

Donde **thingName** se corresponde con el nombre del Objeto al que queremos acceder y **endpointAddress** es la dirección correspondiente al punto de acceso personalizado que podemos obtener en nuestro IoT Core. En el Anexo C: Creación de usuarios y configuración de servicios en AWS, IoT Core se ve de forma detallada como obtener esta dirección. La dirección es la que se muestra en la Figura 45: Punto de acceso personalizado.

Figura 45: Punto de acceso personalizado

Para que la actualización de la Sombra sea posible, creamos una variable llamada **update**, con la información perteneciente al estado "desired"[14] que puede tener la Sombra de un Objeto.

Con el método **updateThingShadow()** actualizaremos la información de la Sombra, incluyendo en esta la información de la variable **update** [34]. El contenido de este se muestra en la Figura 46: Método updateThingShadow.

[14] Como mencionamos en el apartado 5.1.1 Interacción con el Objeto: Thing Shadow, el estado "desired" es el que utilizaremos para almacenar la información perteneciente a la interacción de la aplicación Web, por lo que es este el estado que actualizaremos.

```
var update = {
    "state": {
       "desired" : {
            "dateDesired" : Date.now(),
            "temperatureDesired" : event.temperatureDesired,
            "humidityDesired" : event.humidityDesired,
            "id" : event.id,
            "name" : event.name
        }
    }
};

iotdata.updateThingShadow({payload: JSON.stringify(update),thingName: config.thingName}, function(err,
    if (err) {
        context.fail(err);
    } else {
        console.log(data);
        context.succeed('newStatus: ');
    }
});

```

Figura 46: Método updateThingShadow

Cabe destacar que como con esta función lo que actualizamos es la Sombra de un Objeto, como consecuencia directa, se invoca la función Lambda **shadowToDB,** por lo que también se actualizaría la Base de Datos con la información del nuevo estado de la Sombra.

Por lo tanto, si realizamos una petición POST a la URL que nos proporciona la API Gateway, que veremos en el apartado 5.4 API Gateway, con la información de un nuevo estado de la Sombra, esta se actualizará y podremos consultarlo tanto en el **IoT Core**, como en la **Base de Datos**. Para esta petición hacemos uso de **Postman**. Esto se recoge en las figuras:
Figura 47: Actualización sombra mediante POST usando Postman, Figura 48: Sombra actualizada en IoT Core y Figura 49: Base de Datos actualizada con el nuevo estado de la sombra.

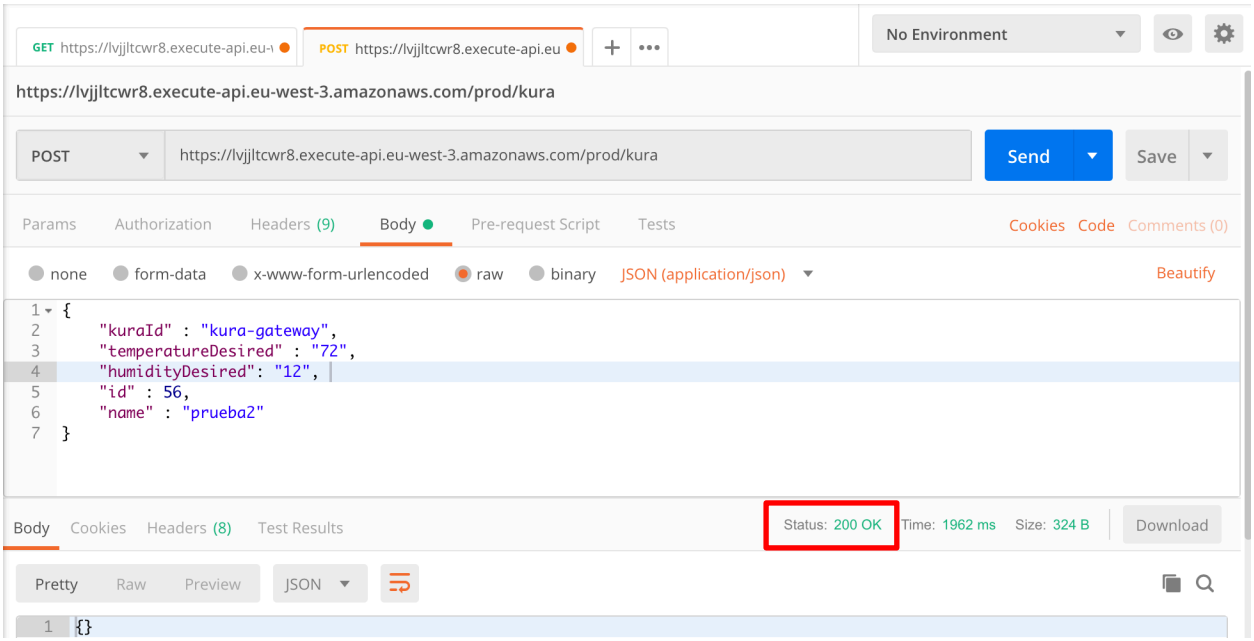


Figura 47: Actualización sombra mediante POST usando Postman

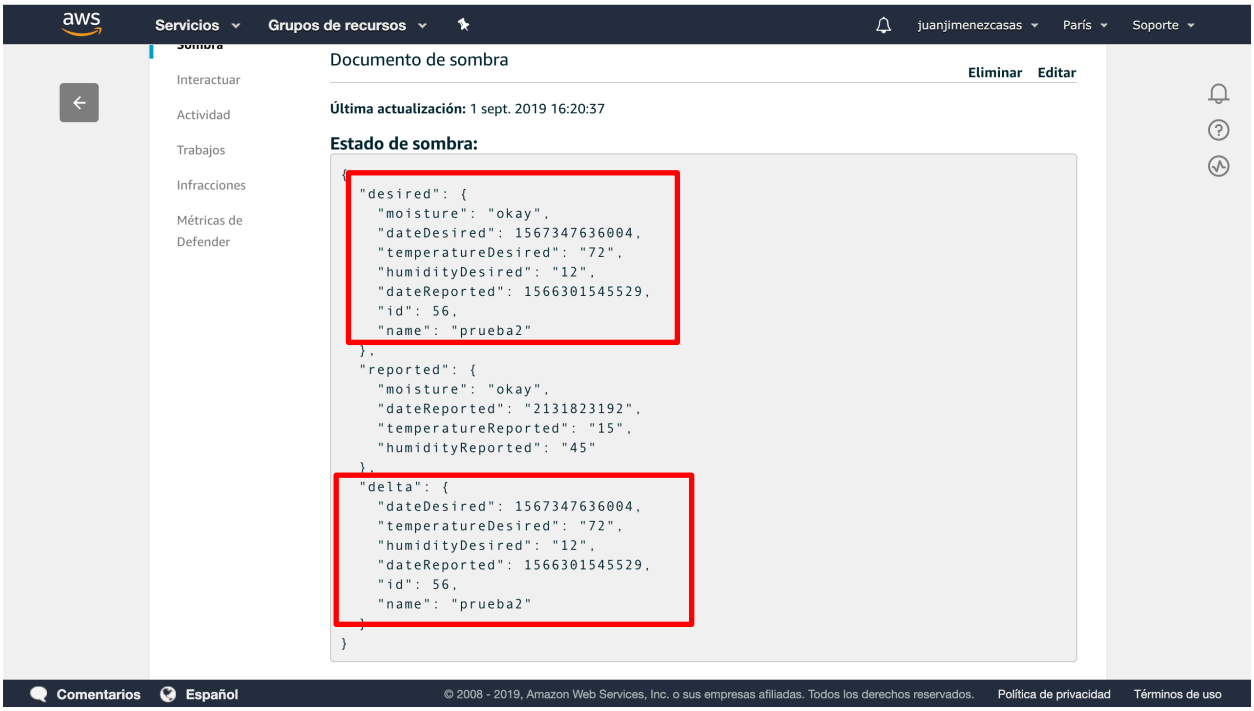


Figura 48: Sombra actualizada en IoT Core

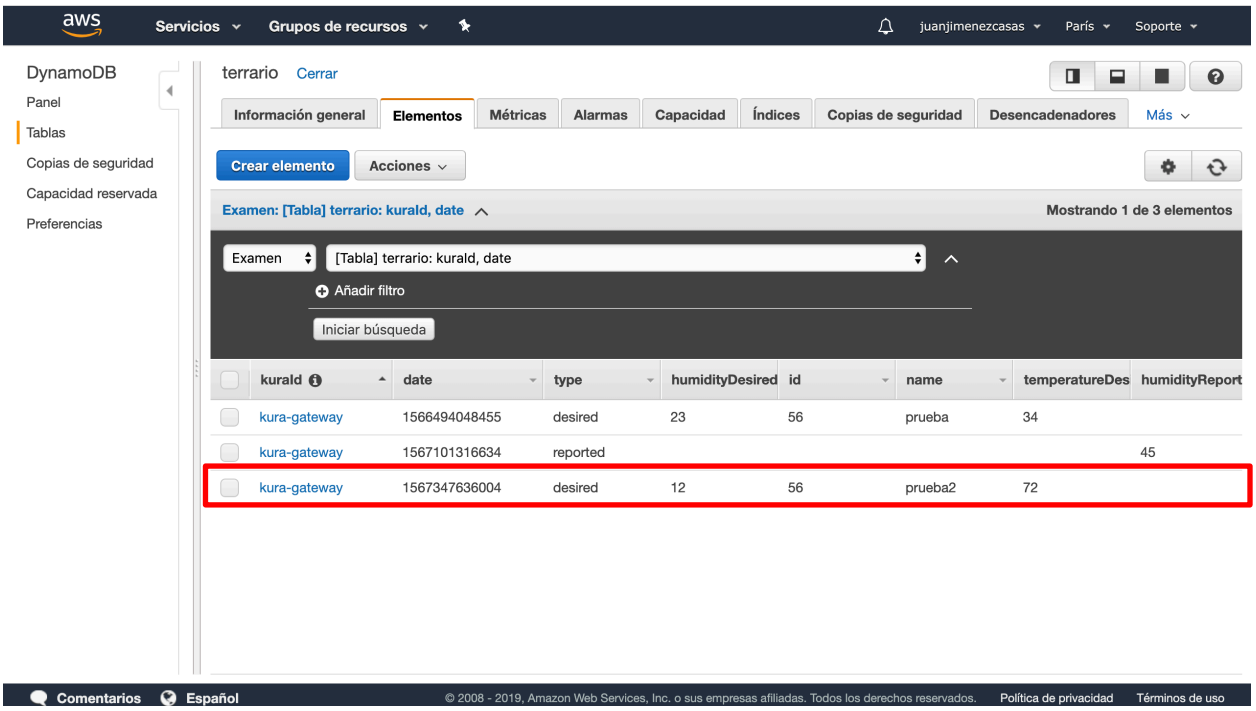


Figura 49: Base de Datos actualizada con el nuevo estado de la sombra.

5.4 API Gateway

Para que nuestro sistema *serverless* en la nube sea de utilidad, es necesario que otras aplicaciones puedan acceder, sin tener conocimiento de cómo estén implementadas estas a través de una API. Amazon AWS propone una herramienta denominada API Gateway para ello. Esta nos proporciona acceso a los recursos que tenemos en AWS a través de una API REST. Por lo tanto, podremos acceder a este servicio realizando peticiones HTTP.

Gracias a esto, tendremos desarrollado el *backend* de nuestra aplicación, sin tener que desplegar nosotros mismos un servidor local, sino que lo "tendremos en la nube". Para que sea más visual lo que se quiere conseguir con todos estos servicios, es de gran ayuda la Figura 50: Backend IoT AWS [35]:

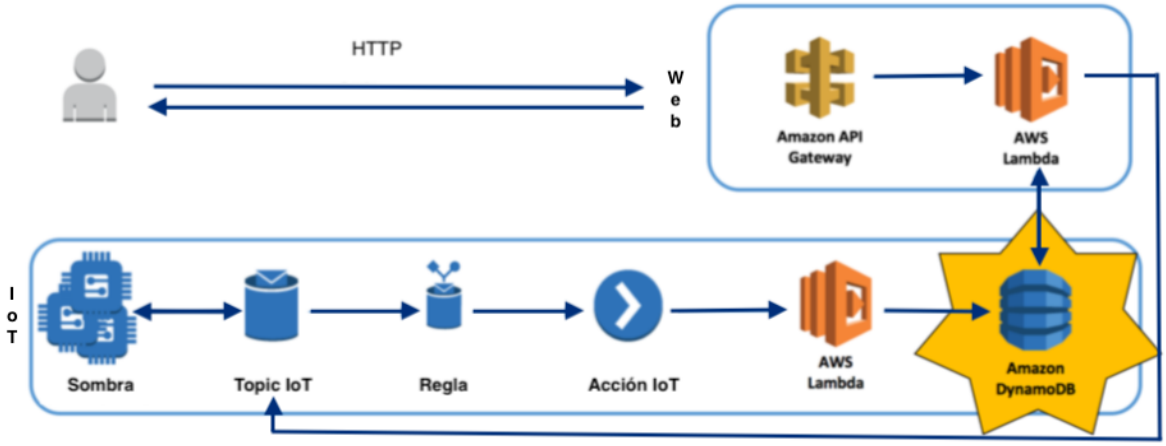


Figura 50: Backend IoT AWS

Como podemos ver, el *backend* de nuestro sistema estaría compuesto por:

- **Entorno IoT:**

El cual involucra la Sombra de un Objeto. Interaccionaremos con esta mediante su **topic IoT**. Además, tendremos una serie de reglas que tienen como objetivo final introducir la información de esta Sombra en una Base de Datos. Para ello se hará uso de una función Lambda llamada **shadowToDB**.

- **Entorno Web:**

En este entorno obtenemos/modificamos la información que hemos almacenado tanto en la Sombra de un Objeto, como en la Base de Datos mediante HTTP. Para ello haremos uso de una API REST, proporcionada por **API Gateway**.

Esta invocará a la función Lambda **dynamodb_read** si queremos leer información de la Base de Datos o a la función **dynamodb_write** si queremos publicar información en el topic IoT para actualizar una sombra, y como consecuencia, actualizar la Base de Datos.

Para ello, en la herramienta API Gateway crearemos una API, llamada **kura**. La información de cómo crear una API se recoge en el Anexo C: Creación de usuarios y configuración de servicios en AWS. En la Figura 51: API kura podemos ver la API creada.

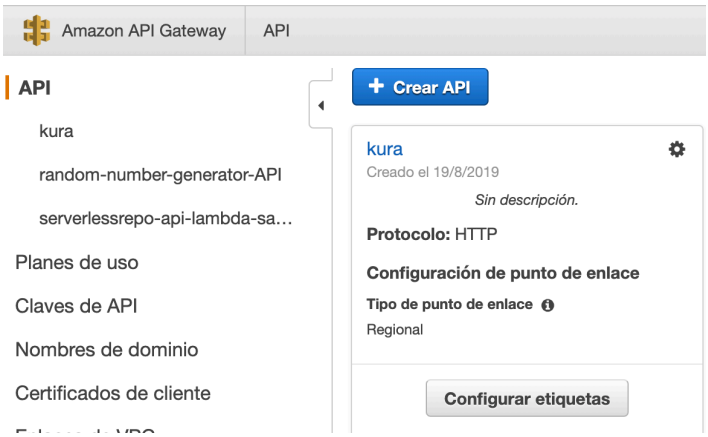


Figura 51: API kura

En esta crearemos los métodos mediante los cuales queremos acceder a los recursos. En nuestro caso serán los métodos GET y POST para obtener y modificar la información respectivamente. En el Anexo C: Creación de usuarios y configuración de servicios en AWS, apartado 7.7 API Gateway se indica como crear los diferentes métodos que se muestran en la Figura 52: Recursos API kura.

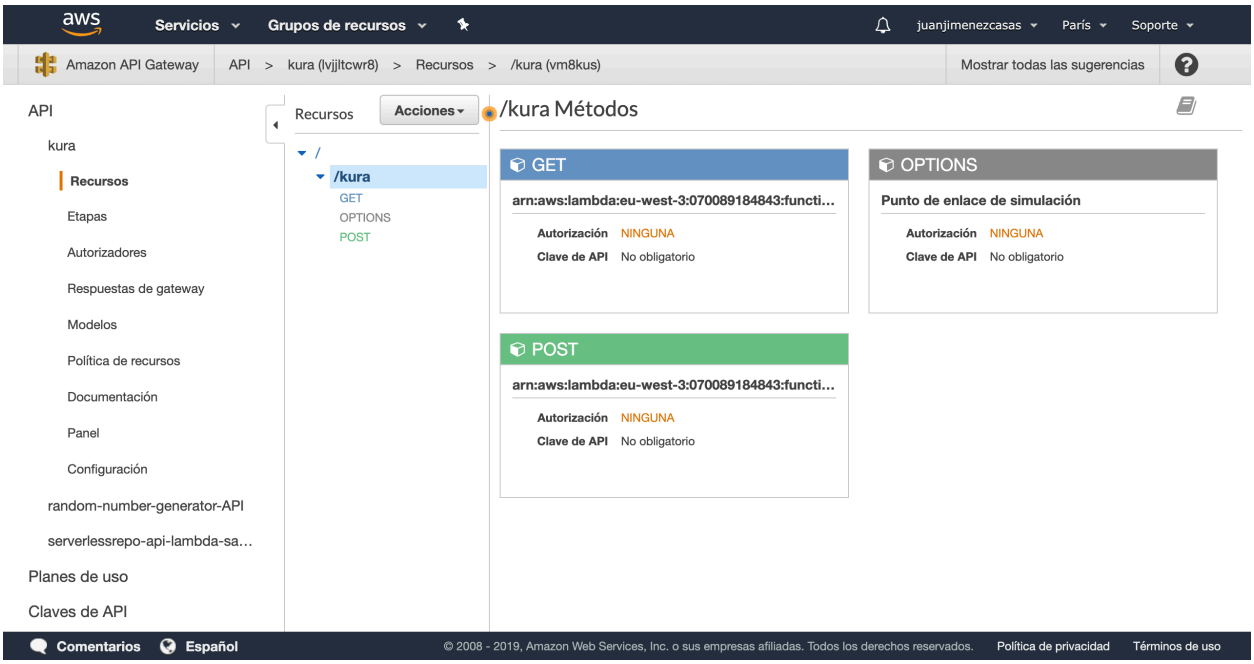


Figura 52: Recursos API kura

5.4.1 Método GET

El método GET correspondiente a la API REST llamada **kura**, que se ha creado con API Gateway, se ha configurado para que llame a una función Lambda. En este caso a la función **dynamodb_read**, vista en el apartado 5.3.2 Función dynamodb_read. Un esquema de este método se muestra en la Figura 53: Método GET. En este se muestra la secuencia en la que se produce la ejecución de un método GET.

No se le han configurado parámetros de seguridad, por lo que actualmente, cualquiera que disponga de la URL que nos proporciona la API al implementarla, podrá acceder a la información de la Base de Datos.

Figura 53: Método GET

Para que podamos acceder a la base de datos y consultar un único elemento de la tabla (utilizar método get () de la función Lambda), es necesario que añadamos una plantilla de mapeo como la que se muestra en la Figura 54: Plantilla de mapeo GET. En el Anexo C: Creación de usuarios y configuración de servicios en AWS se indica como crear tanto un método de la API como las plantillas de mapeo.

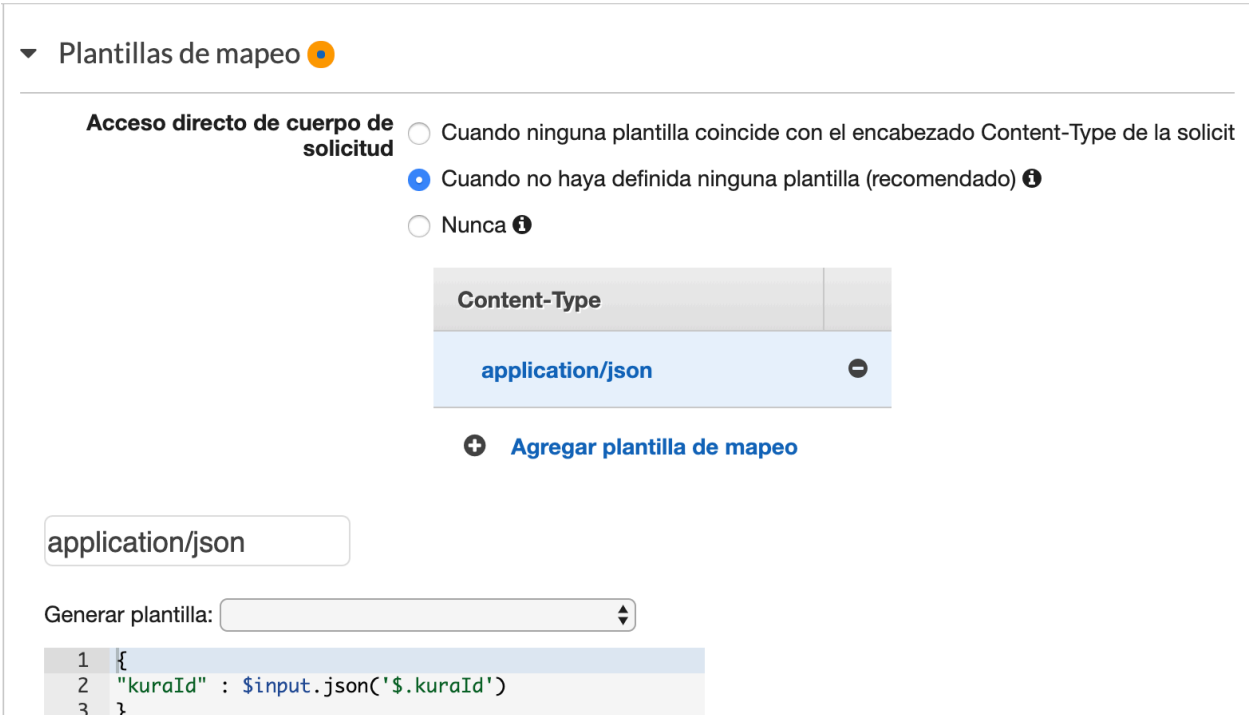


Figura 54: Plantilla de mapeo GET

Esto lo que indica son los distintos parámetros que podemos obtener en la petición GET. En este caso, tenemos un campo en formato JSON: `"kuraId":"identificador"`, cuando enviemos la información a la función Lambda, se corresponderá con la información que contenga `event.kuraId`.

5.4.2 Método POST

El método POST correspondiente a la API REST llamada **kura**, que se ha creado con API Gateway, se ha configurado para que llame a una función Lambda. En este caso a la función **dynamodb_write**, vista en el apartado 5.3.3 Función dynamodb_write. Un esquema de este método se muestra en la Figura 55: Método POST. En este se muestra la secuencia en la que se produce la ejecución de un método POST.

No se le han configurado parámetros de seguridad, por lo que actualmente, cualquiera que disponga de la URL que nos proporciona la API al implementarla, podrá acceder a la información de la Base de Datos.

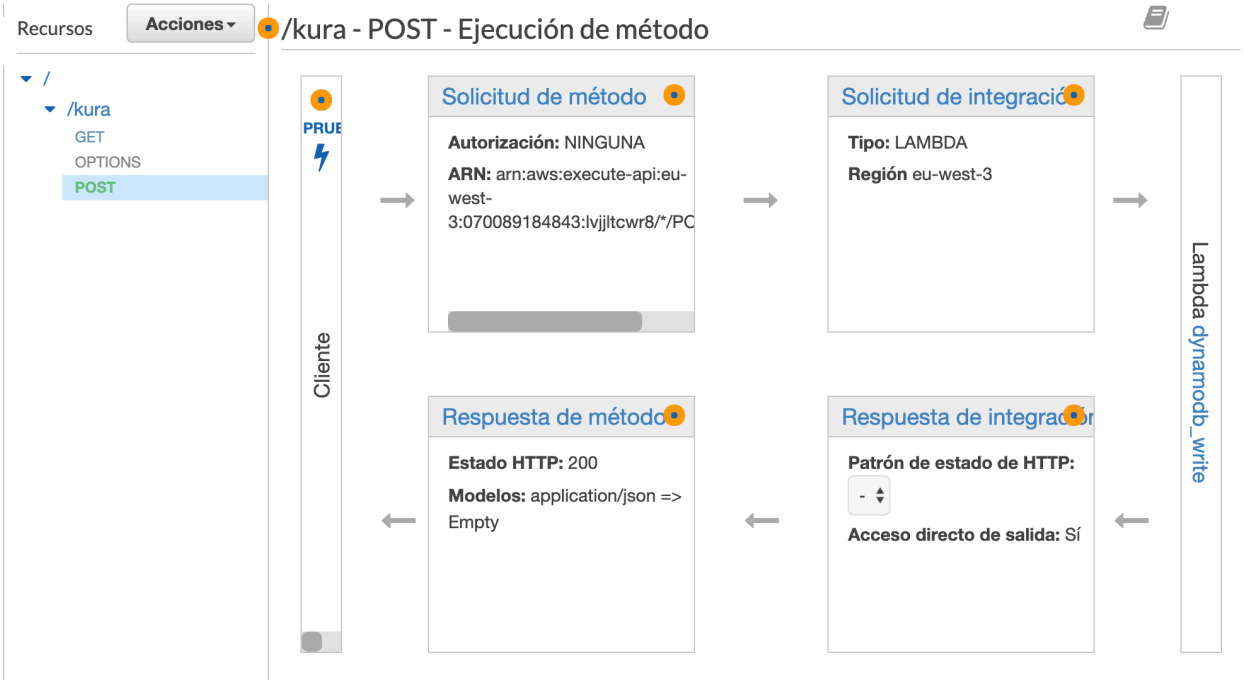


Figura 55: Método POST

Para que podamos actualizar la información de la Sombra, es necesario que añadamos una plantilla de mapeo, para que la nueva información se traslade a la función Lambda tal y como esta lo espera. Esta se muestra en la Figura 56: Plantilla de mapeo POST. En el Anexo C: Creación de usuarios y configuración de servicios en AWS se indica como crear tanto un método de la API como las plantillas de mapeo.

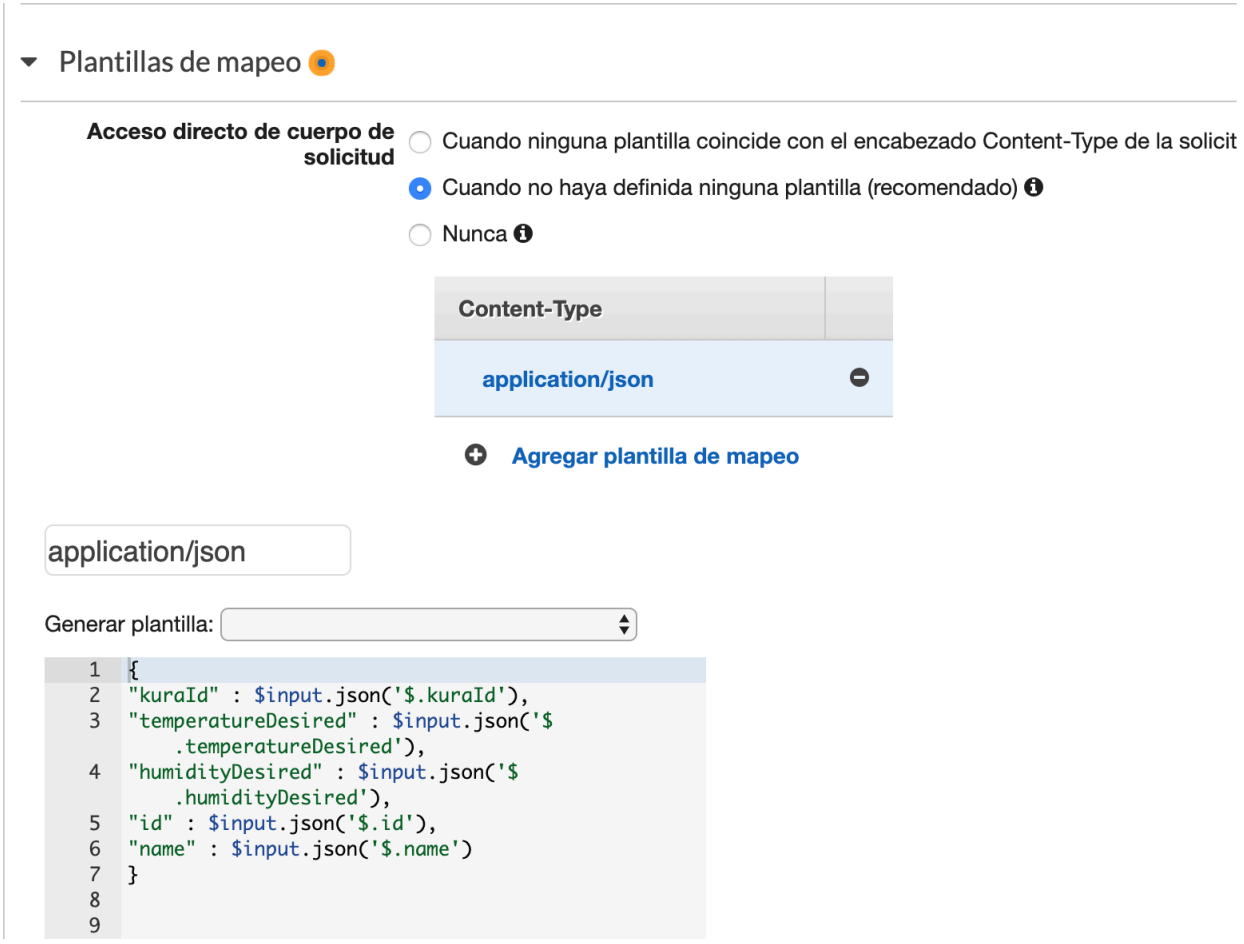


Figura 56: Plantilla de mapeo POST

Esto lo que indica es la información recibida en formato JSON, de igual manera que con el método GET.

5.4.3 Implementación de la API

Una vez hemos creado los distintos métodos que tendrá nuestra API, tendremos que implementarla, para que consigamos una URL a la cual acceder. Para ellos tenemos que crear una **Etapa**. Esto nos permite tener varias API de unos mismos métodos, con el fin de poder probar nuevas versiones de la API sin que la que está en funcionamiento deje de estarlo. En nuestro caso, crearemos la Etapa **prod**. Para más información de cómo crear una Etapa, consultar el Anexo C: Creación de usuarios y configuración de servicios en AWS. En la Figura 57: Etapa prod se muestra la API implementada.

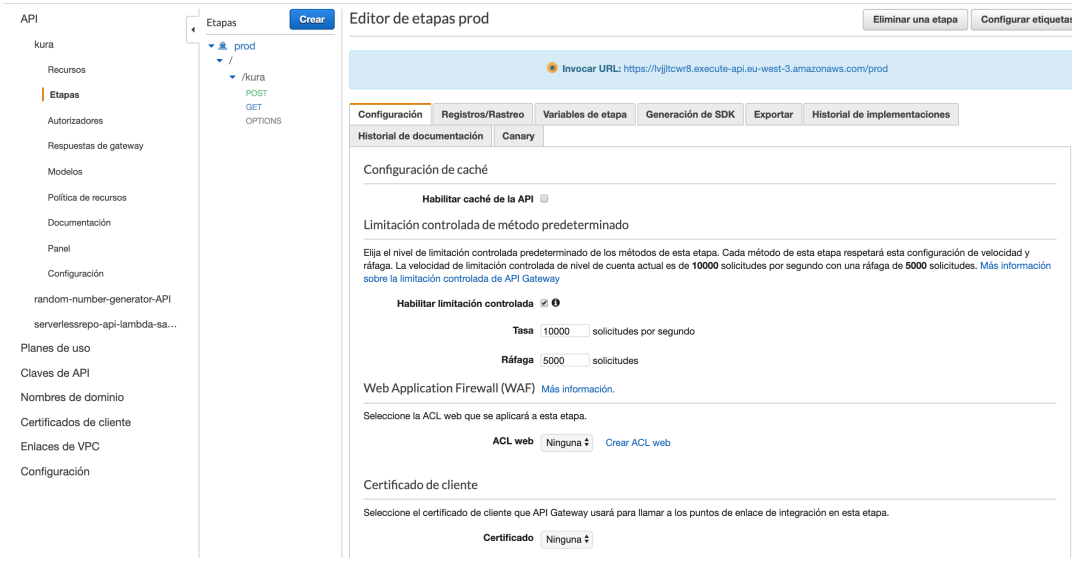


Figura 57: Etapa prod

Si queremos acceder a los recursos que proporciona esta API, tendremos que acceder a la URL

https://lvjjltcwr8.execute-api.eu-west-3.amazonaws.com/prod/kura

El resultado es el que se muestra en la Figura 58: Acceso API. Como podemos ver, la información mostrada se corresponde con el de la base de datos anteriormente mostrada en la Figura 49: Base de Datos actualizada con el nuevo estado de la sombra..

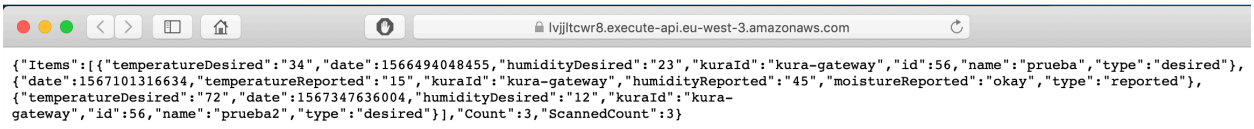


Figura 58: Acceso API

6 APLICACIÓN DESARROLLADA: INTERFAZ WEB ANGULAR

The only way to do great work is to love what you do. If you haven't found it yet, keep looking. Don't settle

- Steve Jobs -

Para poder obtener toda la información almacenada en la Base de Datos, así como poder interactuar con la Sombra de un Objeto y por consecuente, la Raspberry Pi, se ha proporcionado una interfaz de usuario o *frontend* desarrollado en Angular.

6.1 Módulos utilizados

En el fichero **app.module.ts** es donde hemos indicado los distintos módulos de Angular que vamos a utilizar, así como los componentes que hemos declarado nosotros mismos.

app.module.ts

```
1   import { NgModule }       from '@angular/core';
2   import { BrowserModule }  from '@angular/platform-browser';
3   import { FormsModule }    from '@angular/forms';
4   import { HttpClientModule }    from '@angular/common/http';
5   import { AppRoutingModule }     from './app-routing.module';
6   import { AppComponent }         from './app.component';
7   import { DashboardComponent }   from './dashboard/dashboard.component';
8   import { TerrarioDetailComponent }  from './terrario-detail/terrario-detail.component';
9   import { TerrariosComponent }      from './terrarios/terrarios.component';
10  import { TerrarioSearchComponent }  from './terrario-search/terrario-search.component';
11  import { MessagesComponent }    from './messages/messages.component';
12
13  @NgModule({
14    //Modulos que se importan
15    imports: [
16      BrowserModule,
17      FormsModule,
18      AppRoutingModule,
19      HttpClientModule,
20    ],
21    //Modulos que se exportan
22    declarations: [
23      AppComponent,
24      DashboardComponent,
25      TerrariosComponent,
26      TerrarioDetailComponent,
27      MessagesComponent,
28      TerrarioSearchComponent
29    ],
```

Figura 59: Módulo app.module.ts

Como podemos ver en la Figura 59: Módulo app.module.ts, se importan al código los distintos módulos y componentes a través de las rutas donde se encuentran cada uno de los módulos que se quieren referenciar posteriormente en @NgModule{}.

Los módulos que se importan provienen de la librería de Angular y son:

- **BrowserModule:** Permite hacer búsquedas de elementos en los datos de la aplicación.
- **FormsModule:** Permite rellenar un formulario y transferir la información a los datos de la aplicación.
- **AppRoutingModule:** Permite hacer Routing dentro de la aplicación.
- **HttpClientModule:** Permite hacer peticiones HTTP a la API REST.

Los componentes que se declaran y exportan como consecuencia, son los creados para llevar a cabo las funcionalidades deseadas y son:

- **AppComponent:** Componente principal, es la primera vista que se muestra. Se encarga de llamar al resto de componentes.
- **DashboardComponent:** Componente que muestra los últimos terrarios a los que hemos accedido, permitiendo hacer click en ellos para obtener una información más detallada.
- **TerrariosComponent:** Componente que almacena cada uno de los terrarios, así como la información reciente de cada uno de estos.
- **TerrariosDetailComponent:** Componente que contiene la información de cada uno de los terrarios, así como el histórico de temperatura de estos.
- **MessagesComponent:** Componente que nos permite saber el estado de cada una de las peticione realizadas, en forma de mensajes, para la depuración.

La implementación de estos componentes se detalla en el apartado 6.3 Componentes Implementados.

6.2 Servicio Terrario Service

Este servicio es el encargado de realizar las peticiones HTTP a la API REST que nos proporciona API Gateway, vista en el apartado 5.4 API Gateway. Para ello hace uso del módulo **HTTPClient**. Este proporciona los métodos GET y POST necesarios para obtener la información almacenada en AWS.

Los parámetros de configuración que vamos a utilizar son los que se muestran en la Figura 60: Configuración terrario.service.ts.

app.module.ts terrario.service.ts

```
1  import { Injectable } from '@angular/core';
2  import { HttpClient, HttpHeaders } from '@angular/common/http';
3
4  import { Observable, of } from 'rxjs';
5  import { catchError, map, tap } from 'rxjs/operators';
6
7  import { Terrario } from './terrario';
8  import { Items } from './config';
9  import { MessageService } from './message.service';
10
11
12 @Injectable({ providedIn: 'root' })
13 export class TerrarioService {
14
15   private terrariosUrl = 'https://lvjjltcwr8.execute-api.eu-west-3.amazonaws.com/prod/kura';  // URL to web api
16   //private terrariosUrl = 'api/terrarios';
17   //private terrarioUrl = 'https://lvjjltcwr8.execute-api.eu-west-3.amazonaws.com/prod/detail';
18
19   httpOptions = {
20     headers: new HttpHeaders({ 'Content-Type': 'application/json' })
21   };
22
23   constructor(
24     private http: HttpClient,
25     private messageService: MessageService) { }
26
27   /** GET terrarios from the server */
28   getTerrarios (): Observable<Items> {
29   return this.http.get<Items>(this.terrariosUrl)
30     .pipe(
31       tap(data => this.log('fetched heroes'+ JSON.stringify(data))),
32       catchError(this.handleError<Items>('getHeroes',))
33     );
34 }
```

Figura 60: Configuración terrario.service.ts

Para que el resto de los componentes puedan utilizarlo, hay que indicarle que esa clase está disponible para ser inyectada como dependencia. Esto se consigue con el comando:

```
@Injectable ({providedIn : 'root'})
```

Cabe destacar que `providedIn` indica los inyectores que proporcionarán el inyectable (en este caso Terrario Service). Por defecto, el inyector es `'root'` para toda la aplicación.

Crearemos una variable que contenga la ruta a la que vamos a hacer las peticiones HTTP. Esta es la que nos ha proporcionado API Gateway. Es la que se muestra en el apartado 5.4.3.

```
private terrariosUrl =
'https://lvjjltcwr8.execute-api.eu-west- 3.amazonaws.com/prod/kura';
```

Para que el contenido que enviemos en las cabeceras pueda ser interpretado como espera API Gateway, enviaremos la información en formato JSON. Eso lo conseguimos con las líneas 19-21 de la Figura 60: Configuración terrario.service.ts, haciendo uso de las propiedades `httpOptions` de `HttpClient`.

Por último, en el constructor crearemos un objeto de tipo HttpClient, que lo referenciaremos para que nos proporcione los métodos HTTP, y otro de tipo MessageService, para que podamos ver la información a la hora de depurarlo.

6.2.1 Métodos implementados

Todos los métodos de HttpClient devuelven un RxJs Observable. Eso se representa mediante Observable<Tipo>. En nuestro caso, las peticiones HTTP devuelven/reciben un JSON con el formato de la interfaz desarrollada **Items**. La interfaz Items contiene los siguientes atributos:

```
import { Terrario } from './terrario';

export interface Items {
  items: Terrario[];
  count: number;
  scannedCount: number;
}
```

Por lo tanto, en el atributo items de un objeto de tipo Items tendremos una tabla de objetos de tipo Terrario, que contiene la información de un terrario en un momento dado.

- **getTerrarios() Observable<Items>:** Este método nos permite obtener un elemento de tipo Items. Este contendrá la información relativa a todos los terrarios. Para poder depurar en caso de error, introduciremos en una tubería el contenido del observable, esto se consigue con el `.pipe()`. El contenido de este método se muestra en la Figura 61: Método getTerrarios.

```
/** GET terrarios from the server */
getTerrarios (): Observable<Items> {
return this.http.get<Items>(this.terrariosUrl)
  .pipe(
    tap(data => this.log('fetched terrarios'+ JSON.stringify(data))),
    catchError(this.handleError<Items>('getTerrarios',))
  );
}
```

Figura 61: Método getTerrarios

- **getTerrario (kuraId : string) Observable<Items>:** Este método nos permite obtener un elemento de tipo Items. Este contendrá la información relativa de los terrarios con identificador **kuraId** coincidente con el pasado como parámetro. Se utilizará cuando se quiera obtener la información de un terrario únicamente. Para poder depurar en caso de error, introduciremos en una tubería el contenido del observable, esto se consigue con el `.pipe()`. El contenido de este método se muestra en la Figura 62: Método getTerrario.

```
42   //GET Terrario from the server
43   getTerrario (kuraId:string): Observable<Items> {
44     return this.http.get<Items>(this.terrariosUrl+'/?kuraId='+kuraId)
45       .pipe(
46         tap(data => this.log('fetched terrario'+ JSON.stringify(data))),
47         catchError(this.handleError<Items>('getTerrario',))
48       );
49   }
```

Figura 62: Método getTerrario

- **addTerrario (terrario : Posting): Observable <Terrario>:** Este método nos permite enviar a la API REST informacón perteneciente a un terrario. Lo utilizaremos para controlar el terrario desde la aplicación Web. Se ha creado una interfaz denominada Posting que contendrá la información que vamos a enviar en el cuerpo de la petición POST que este realiza. La información se obtiene a través del formulario que se encuentra en el componente 6.3.5 Componente Terrarios Detail. El contenido de este método se muestra en la Figura 63: Método addTerrario.

```
104
105   /** POST: add a new terrario to the server */
106   addTerrario (terrario: Posting): Observable<Terrario> {
107     console.log(terrario);
108     return this.http.post<Terrario>(this.terrariosUrl, terrario, this.httpOptions).pipe(
109       tap((newTerrario: Terrario) => this.log(`added terrario w/ id=${newTerrario.id}`)),
110       catchError(this.handleError<Terrario>('addTerrario'))
111     );
112   }
```

Figura 63: Método addTerrario

6.3 Componentes Implementados

Los componentes implementados harán uso de las interfaces creadas para representar a los objetos recibidos en el JSON. Se hará uso de la Interfaz **Items**, mencionada anteriormente en el apartado 6.2.1 y de la Clase **Terrario**, que representa a un terrario, teniendo como atributos los siguientes:

```
export class Terrario {
  id: number;
  date: number;
  name: string;
  temperatureReported: string;
  temperatureDesired: string;
  kuraId: string;
  humidityReported: string;
  humidityDesired: string;
}
```

6.3.1 Componente Routing

En este componente indicaremos las distintas rutas a las que podemos acceder dependiendo de la dirección URL en la que nos encontremos. Para conseguir esto, se debe crear la clase **approuting.module.ts** con el contenido que se muestra en la
Figura 64: app-routing.module.ts.

dashboard.component.ts app-routing.module.ts ×

```
1  import { NgModule }             from '@angular/core';
2  import { RouterModule, Routes } from '@angular/router';
3
4  import { DashboardComponent }   from './dashboard/dashboard.component';
5  import { TerrariosComponent }      from './terrarios/terrarios.component';
6  import { TerrarioDetailComponent }  from './terrario-detail/terrario-detail.component';
7
8  const routes: Routes = [
9    { path: '', redirectTo: '/dashboard', pathMatch: 'full' },
10   { path: 'dashboard', component: DashboardComponent },
11   { path: 'detail/:kuraId', component: TerrarioDetailComponent },
12   { path: 'terrarios', component: TerrariosComponent }
13 ];
14
15 @NgModule({
16   imports: [ RouterModule.forRoot(routes) ],
17   exports: [ RouterModule ]
18 })
19 export class AppRoutingModule {}
20
```

Figura 64: app-routing.module.ts

Lo destacable de esta clase es la redirección que se hace al componente **Dashboard**, al acceder a la página inicial. De esta forma, en la página inicial se nos debe mostrar el Dashboard también. Se corresponde con la línea 9 de la
Figura 64: app-routing.module.ts.

6.3.2 Componente principal App Component

Este componente es el primero que se muestra al acceder a la aplicación Web. Además, sirve como enlace para acceder al resto de componentes.

6.3.2.1 Vista HTML

app.component.html ×

```
1  <h1>{{title}}</h1>
2  <nav>
3    <a routerLink="/dashboard">Dashboard</a>
4    <a routerLink="/terrarios">Terrarios</a>
5  </nav>
6  <router-outlet></router-outlet>
7  <app-messages></app-messages>
```

Figura 65: app.component.html

Como podemos ver en la Figura 65: app.component.html, el título, correspondiente a `{{title}}` se cargará de forma dinámica, dependiendo del contenido de la variable **title** en la clase TypeScript. Esto se consigue gracias a las directivas que se representan como: `{{variable}}`.

Posteriormente, como podemos ver en las líneas 4 y 5 se añade una barra de navegación con dos iconos, que enlazan con los componentes Dashboard y Terrarios.

Con las líneas 6 y 7 se hace referencia a dos componentes: el componente propio de la ruta (`router-outlet`) y el de mensajes (`app-messages`). Estos se mostrarán en la parte más inferior de la página principal.

Por lo tanto, la vista de esta desde el navegador se ve cómo podemos apreciar en la Figura 66: Página principal.

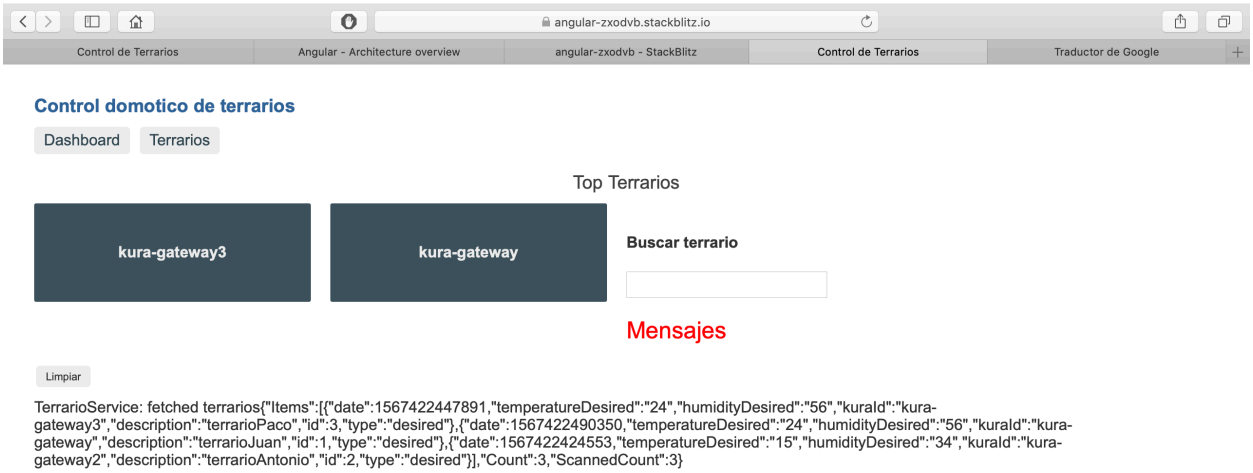


Figura 66: Página principal

6.3.2.2 Datos y lógica TypeScript

Para que todo esto sea posible, al crear una clase debemos indicar mediante @Component () tres propiedades:

- **selector**: Se corresponde con el identificador de este componente para ser referenciado desde otras vistas.
- **templateUrl**: Correspondiente a la vista de este componente.
- **styleUrls**: Correspondiente a los estilos de la vista de este componente.

Posteriormente, exportamos la clase creada y asignamos un título para que la vista pueda cargarlo. Todo esto se ve referenciado en la Figura 67: app.component.ts.

app.component.ts

```
1  import { Component } from '@angular/core';
2
3  @Component({
4    selector: 'app-root',
5    templateUrl: './app.component.html',
6    styleUrls: ['./app.component.css']
7  })
8  export class AppComponent {
9    title = 'Control domotico de terrarios';
10 }
11
```

Figura 67: app.component.ts

6.3.3 Componente Terrarios Component

Este componente es el que utilizaremos para obtener y asignar a la clase terrario la información obtenida de la API Gateway.

6.3.3.1 Vista HTML

Este componente nos mostrará en forma de lista el nombre de cada uno de los terrarios, así como su identificador. Además, se podrá hacer click en cada uno de los terrarios y nos enviará al componente terrarios-detail de este. La vista en el navegador Web sería la mostrada en la Figura 68: Vista Terrarios Component.

Figura 68: Vista Terrarios Component

Para representar cada uno de los terrarios en forma de botones se ha empleado el siguiente código:

```
<ul class="terrarios">
  <li *ngFor="let terrario of terrarios">
    <a routerLink="/detail/{{terrario.kuraId}}">
      <span class="badge">{{terrario.id}}</span>{{terrario.kuraId}}
    </a>
    <button class="delete" title="delete terrario"
      (click)="delete(terrario)">x</button>
  </li>
</ul>
```

Lo destacable de este código es utilizar los bucles `*ngFor`. Estos nos permiten recorrer la tabla terrarios obtenida del modelo de datos y asignarla a un elemento terrario en la vista para poder imprimir todos los elementos de esa tabla. Por lo tanto, en la lista que se crea, el elemento `terrario.id` y `terrario.kuraId` se corresponderá con el elemento que se esté obteniendo en ese momento con el bucle `*ngFor`.

6.3.3.2 Datos y lógica TypeScript

Para crear la clase Terrarios Component y que esta sea accesible desde el resto de la Web, definimos su componente como:

```
@Component({
  selector: 'app-terrarios',
  templateUrl: './terrarios.component.html',
  styleUrls: ['./terrarios.component.css']
})
```

Creamos dos objetos, para hacer el mapeo del JSON obtenido a clases a las que podamos acceder.

```
  terrarios: Terrario[];
  items: Items;
```

Por último, cabe destacar el método que se llamará para obtener los terrarios. Este se suscribirá a el método getTerrarios() de TerrarioService.

```
  getTerrarios(): void {this.terrarioService.getTerrarios()
.subscribe(items => {
      this.items = {
        items: (items as any).Items,
        count: (items as any).Count,
        scannedCount: (items as any).ScannedCount
      };
      this.terrarios = this.items.items;
      console.log(this.terrarios);
    });
  }
```

Esto lo que hace es asignar al objeto `items`, el JSON obtenido de la petición GET. Posteriormente, al objeto `terrarios` se le asignan todos los terrarios que se obtienen, que es el contenido de la variable `items` del objeto `items`. De esta forma, la vista obtiene todos los terrarios.

6.3.4 Componente Dashboard Component

Este componente nos permite obtener una vista rápida de los últimos terrarios a los que hemos accedido.

6.3.4.1 Vista HTML

En este componente, la vista queda tal y como se muestra en la Figura 66: Página principal. La única diferencia es el tamaño de los botones.

6.3.4.2 Datos y lógica TypeScript

La lógica de este componente es idéntica a la del componente Terrarios, pero a la hora de asignar los terrarios al objeto terrarios, sólo se asignarán los 2 primeros. Esto se consigue con la siguiente línea de código:

```
this.terrarios = this.items.items.slice(0,2);
```

6.3.5 Componente Terrarios Detail

Este componente es el que nos permite ver el histórico de temperaturas que ha tenido un terrario dado. Además, permite controlar el terrario mediante un formulario.

6.3.5.1 Vista HTML

Este componente nos mostrará en forma de lista el histórico de temperatura y humedad de un terrario.

Además, se podrá rellenar un formulario que nos permita activar o desactivar el controlador y enviar la temperatura o humedad deseada. La vista en el navegador Web sería la mostrada en la Figura 69: Vista Terrarios Detail.

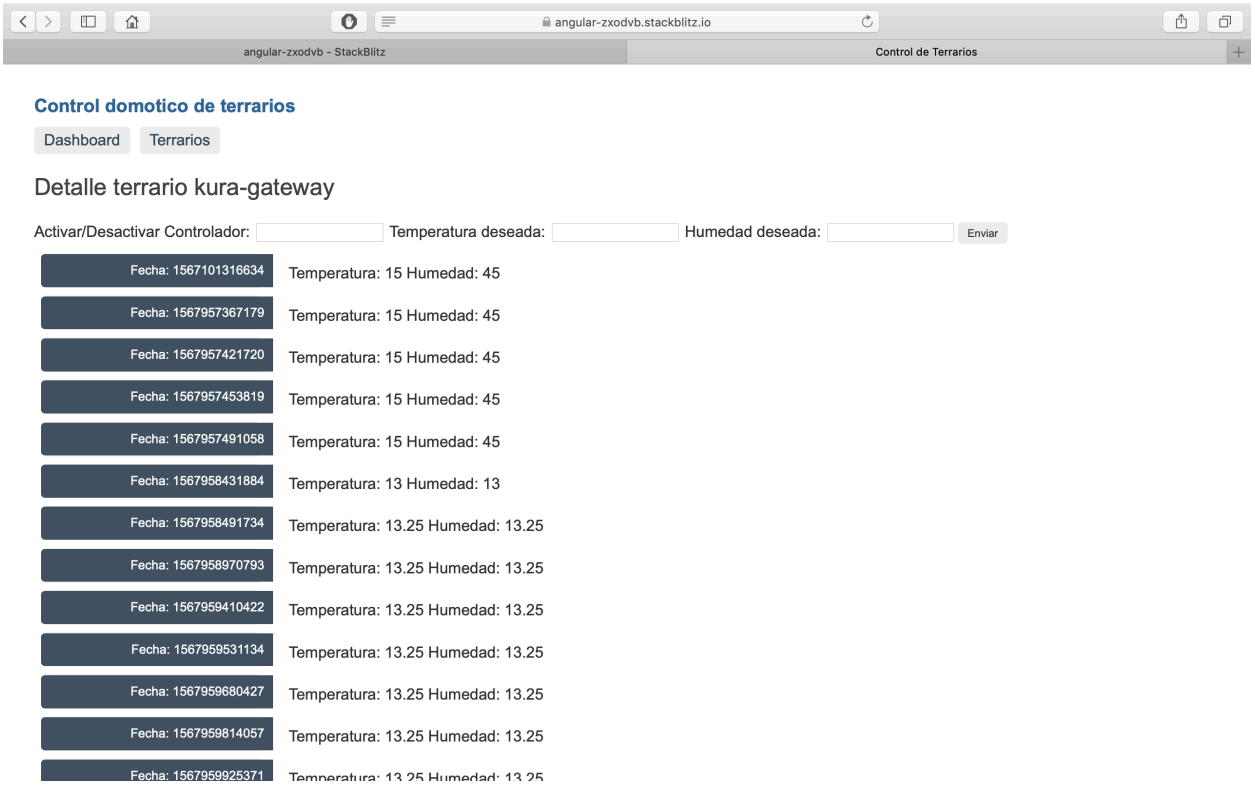


Figura 69: Vista Terrarios Detail

Para representar cada uno de los estados del terrario en forma de lista se ha empleado el siguiente código:

```
<ul class="terrarios">
    <li *ngFor="let terrario of terrarios">
      <a>
        <span class="badge">Fecha: {{terrario.date}} </span>
Temperatura: {{terrario.temperatureReported}} Humedad:
{{terrario.humidityReported}}
      </a>
    </li>
  </ul>
```

Para crear el formulario y asignar esta información a los datos de la parte TypeScript se ha empleado el siguiente código:

```
  <div>
    <label>Activar/Desactivar Controlador:
      <input #terrarioControl />
    </label>
    <label> Temperatura deseada:
      <input #temperature />
    </label>
```

```
    <label> Humedad deseada:
      <input #humidity />
    </label>
    <!-- (click) passes input value to add() and then clears the input
-->
    <button (click)="add(terrarioControl.value, temperature.value,
humidity.value); terrarioControl.value=''; temperature.value='';
humidity.value=''">
      Enviar
    </button>
  </div>
```

De esta forma, cuando le demos al botón enviar de la interfaz Web, se enviará una petición POST a la URL "https://lvjjltcwr8.execute-api.eu-west-3.amazonaws.com/prod/kura" proporcionada por API Gateway de AWS con el contenido del formulario en el cuerpo de la petición POST.

Esta se encargará de invocar a la función Lambda 5.3.3 Función dynamodb_write para actualizar la Sombra del Objeto correspondiente y la Base de Datos. Como la Raspberry Pi está suscrita a esta mediante el CloudSuscriber visto en el apartado 4.2.2 CloudSuscriber, esta recibirá la información de control y se encenderán/apagarán los LEDs tal y como se muestra en las figuras: Figura 26: Control enciende LED, Figura 27: Control apaga LED del apartado 4.4 Kura Wires y GPIODriver: Control de LEDs.

6.3.5.2 Datos y lógica TypeScript

Este compontente se ha definido utilizando la dependencia:

```
@Component({
  selector: 'app-terrario-detail',
  templateUrl: './terrario-detail.component.html',
  styleUrls: [ './terrario-detail.component.css' ]
})
```

Como queremos que la información de ruta sea utilizable por RoutingComponent, así como poder obtener información de la ruta para hacer la petición GET necesaria, necesitamos implementar el siguiente constructor:

```
  constructor(
    private route: ActivatedRoute,
    private terrarioService: TerrarioService,
  ) {}
```

Con esto conseguimos introducir en la variable `route`, la URL en la que nos encontramos.
Para obtener la variable `kuraId`, que es la clave de un terrario, sólo tenemos que obtenerla de la variable route:

```
let kuraId = this.route.snapshot.paramMap.get('kuraId');
```

Para obtener la información del terrario, llamamos al método `getTerrario(kuraId)` del apartado 6.2 Servicio Terrario Service, en la que nos devuelve la información y la asignamos de la misma forma que en TerrariosComponent, pero en este caso, sólo es la información relativa a un terrario.

Para enviar la información de control, llamamos al método `addTerrario(terrarioEnv)` del apartado 6.2 Servicio Terrario Service.

Para que esta petición se produzca de forma satisfactoria, hemos tenido que crear una interfaz denominada Posting. La interfaz Posting contiene los atributos:

```
export class Posting {

  id: number;
  name: string;
  temperatureDesired: string;
  kuraId: string;
  humidityDesired: string;
  control: boolean;

}
```

Esta nos permite introducir información de control en un terrario y de temperatura/humedad deseada. Es la que se utiliza para mapear la información procedente del formulario. Esto se consigue con el siguiente código:

```
add(control: boolean,temperature: string, humidity: string): void {

    this.terrarioEnv = {
      id : this.terrario.id,
      temperatureDesired :temperature,
      kuraId : this.kura,
      humidityDesired : humidity,
      name: this.terrario.name,
      control : control,
    }
}
```

Siendo el método `add` el que se invoca cuando se pulsa el botón **Enviar** del formulario.

6.3.6 Componente Messages

Este componente nos permite visualizar información de depuración en la pantalla. Para definirlo se ha utilizado la siguiente dependencia:

```
@Component({
  selector: 'app-messages',
  templateUrl: './messages.component.html',
  styleUrls: ['./messages.component.css']
})
```

Por lo tanto, en el resto de las vistas se incluirá `<app-messages></app-messages>` para que se muestre la información capturada en este. Muestra la información como se puede apreciar en las figuras: Figura 66: Página principal y Figura 68: Vista Terrarios Component.

Con la finalidad de aclarar el funcionamiento y las posibles interacciones que esta aplicación Web ofrece, en la Figura 70: Casos de uso se muestran los distintos casos de uso que esta aplicación proporciona.

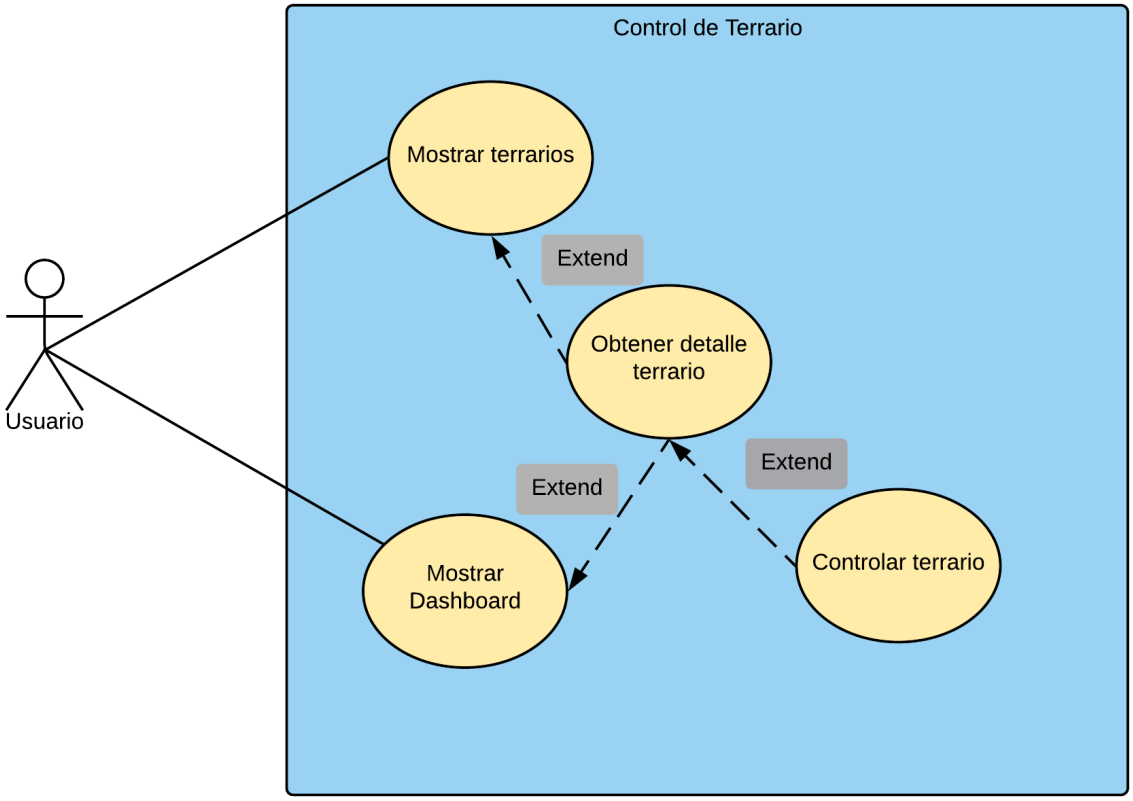


Figura 70: Casos de uso

Además en la Figura 71: Diagrama de secuencia obtener terrario, se muestra un diagrama de secuencia para el caso de uso de obtener el detalle de un terrario.

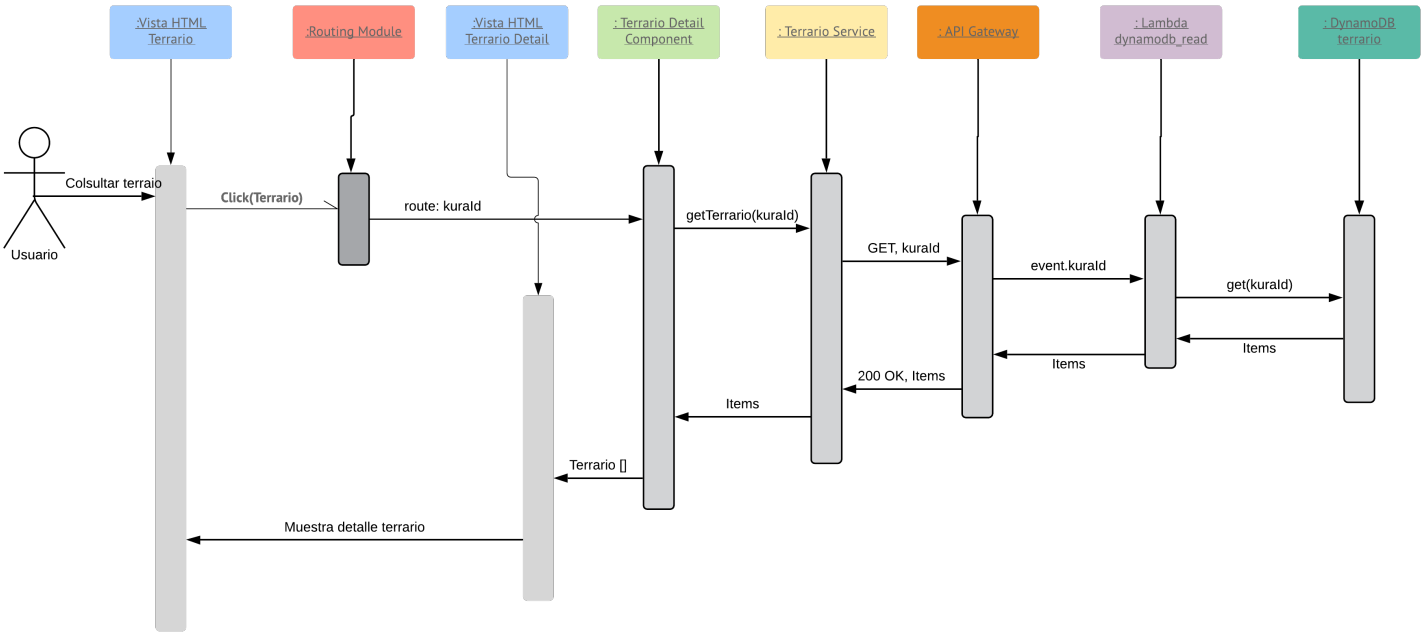


Figura 71: Diagrama de secuencia obtener terrario

Por último, en la Figura 72: Diagrama de secuencia controlar terrario se muestra un diagrama de secuencia para el caso de uso de controlar terrario.

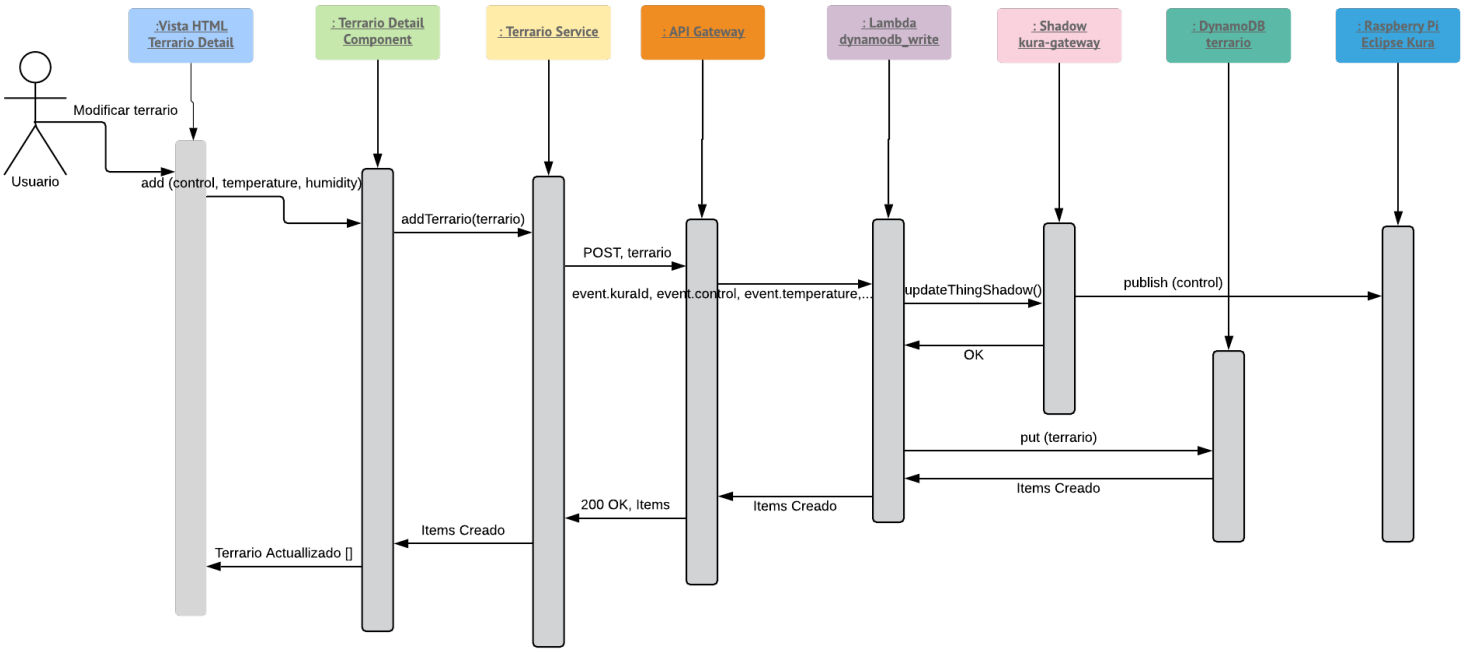


Figura 72: Diagrama de secuencia controlar terrario

7 CONCLUSIONES Y LÍNEAS FUTURAS

If you get up in the morning and think the future is going to be better, it is a bright day. Otherwise, it´s not.

- Elon Musk-

Como punto final a este Trabajo de Fin de Grado me gustaría reflexionar sobre los conocimientos adquiridos, la funcionalidad conseguida para este, así como la capacidad de mejora que tiene.

Desde el punto de vista del aprendizaje, me gustaría remarcar los conocimientos adquiridos para la creación de un servicio basado en el Cloud Computing, que nos proporciona facilidades como no preocuparse por el mantenimiento, por el aprovisionamiento de los recursos, por la integridad de la información. Además, todo el desarrollo modular con Eclipse Kura y la Raspberry Pi me han hecho tener consciencia de la importancia de la modularidad y la reutilización de código, aspectos clave hoy en día para el desarrollo.

Desde el punto de vista de la funcionalidad desarrollada, considero que se obtiene una buena solución inicial para el problema que se planteaba. Sin embargo, aún tiene mucho margen de mejora. Actualmente, se proporciona un sistema IoT, que permite la monitorización de la temperatura y humedad de un terrario y el control de unos LEDs mediante una interfaz Web.

La interfaz Web implementada en Angular, es capaz de mostrar la información y acceder a los distintos recursos de una forma sencilla. Por contraposición, la forma de mostrar esta información no es la más visual ni intuitiva, ya que lo mejor para esto sería hacer uso de gráficas y estilos a la orden del día. Para ello considero una mejora sustancial el incluir una plantilla de estilos en Bootstrap.

En el apartado del Cloud Computing, se otorga al sistema de una seguridad interna en el procesamiento y gestión de la información, así como en la comunicación con la Raspberry Pi. Se han utilizado diferentes servicios dentro de este entorno, como son IoT Core, DynamoDB como Base de Datos, Lambda y API Gateway. Sin embargo, la API REST no proporciona ningún tipo de seguridad, por lo que cualquiera que conozca la URL de esta, puede acceder a los recursos mediante los métodos que propone. Por lo tanto, en un futuro estaría a bien incorporarle esa seguridad que le falta.

Finalmente, la implementación en Eclipse Kura ha sido la que más quebraderos de cabeza me ha dado, aunque a priori parecía la más sencilla. Esto se debe a que se necesita tener un conocimiento muy profundo de cómo funciona para poder empezar a desarrollar tus propias aplicaciones. Una vez se adquiere, la modularidad y la cantidad de recursos que proporciona "lo ponen todo en bandeja". A pesar de todo, me hubiese gustado implementar más funcionalidades en esta, como la integración con motores y relés que se accionen en función de una temperatura y humedad deseada.

ANEXO A: INSTALACIÓN DE SENSORES DHT11 Y LEDS EN RASPBERRY PI

En este anexo se explica paso a paso cómo conectar los sensores de humedad y temperatura DHT11 a los pines GPIO de la Raspberry Pi, para que se pueda obtener la información de estos. Además, se muestra el conexionado de los LEDs utilizados.

Para el conexionado del sensor DHT11, se ha utilizado la siguiente referencia [2]. Inicialmente, hay que identificar qué tipo de sensor DHT11 tenemos. Los hay de 3 pines y de 4 pines. La diferencia entre estos radica en que el de 3 pines no necesita una resistencia para conectarlo con los pines GPIO de la Raspberry Pi, ya que la trae incorporada. En el caso de que tengamos uno de 4 pines, necesitaremos adicionalmente una resistencia de entre 4,7K y 10 K Ohmios que colocaremos entre el pin 1 y el in 2 [2]. En nuestro caso, el sensor es de 3 pines, por lo que no la necesitaremos.

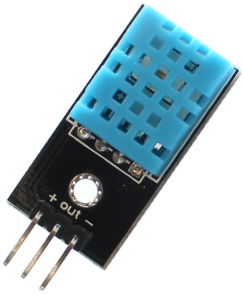


Figura 73: Sensor DHT11

Para hacer referencia a los pines del sensor mostrados en la Figura 73: Sensor DHT11, de izquierda a derecha nos referiremos como:

- Pin 1: Positivo.
- Pin 2: Datos.
- Pin 3: Negativo.

Para referirnos a los pines GPIO de la Raspberry Pi los numeraremos como se indica en la Figura 74: Pines GPIO.

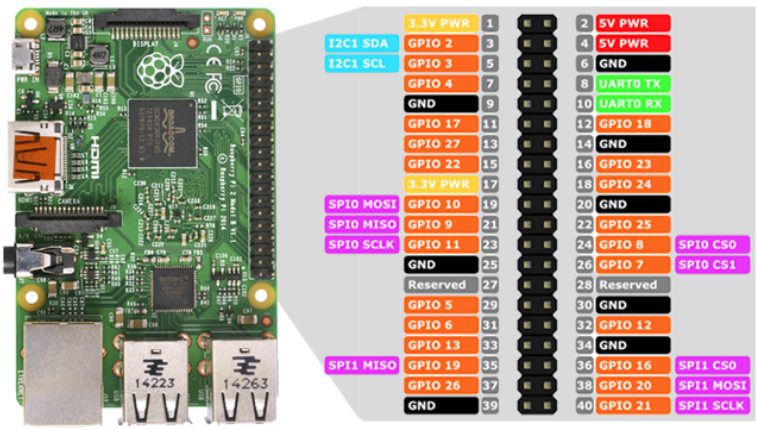


Figura 74: Pines GPIO

Por lo tanto, el conexionado del Pin 1 debería hacerse con un pin de voltaje positivo, por ejemplo, el Pin 1 (3.3V) de la Raspberry Pi.

El conexionado del Pin 3 deberá hacerse con un pin de tierra, en este caso el Pin 39 (GND) de la Raspberry Pi.

Para la transmisión de datos, debemos conectar el Pin 2 con alguno de los pines denominados como GPIO XX, que son los que permiten el intercambio de datos. En nuestro caso, lo haremos con el Pin 38 (GPIO 20).

Con todo esto, si ejecutamos el programa que nos permite leer la información de los sensores DHT11 (**AdafruitDHT.py**) nos mostrará la siguiente salida:

```
pi@raspberrypi:~/Adafruit_Python_DHT/examples $ python AdafruitDHT.py 11 20
Temp=29.0*  Humidity=45.0%
```

Para el conexionado de los LEDs, se han conectado directamente al PIN GPIO del que van a obtener información en el polo positivo de este. El polo negativo se ha conectado a una resistencia de 1K Ohmios y posteriormente a un pin de tierra (GND). El conexionado tanto de los sensores como del LED se muestra en la Figura 75: Conexionado Raspberry Pi.

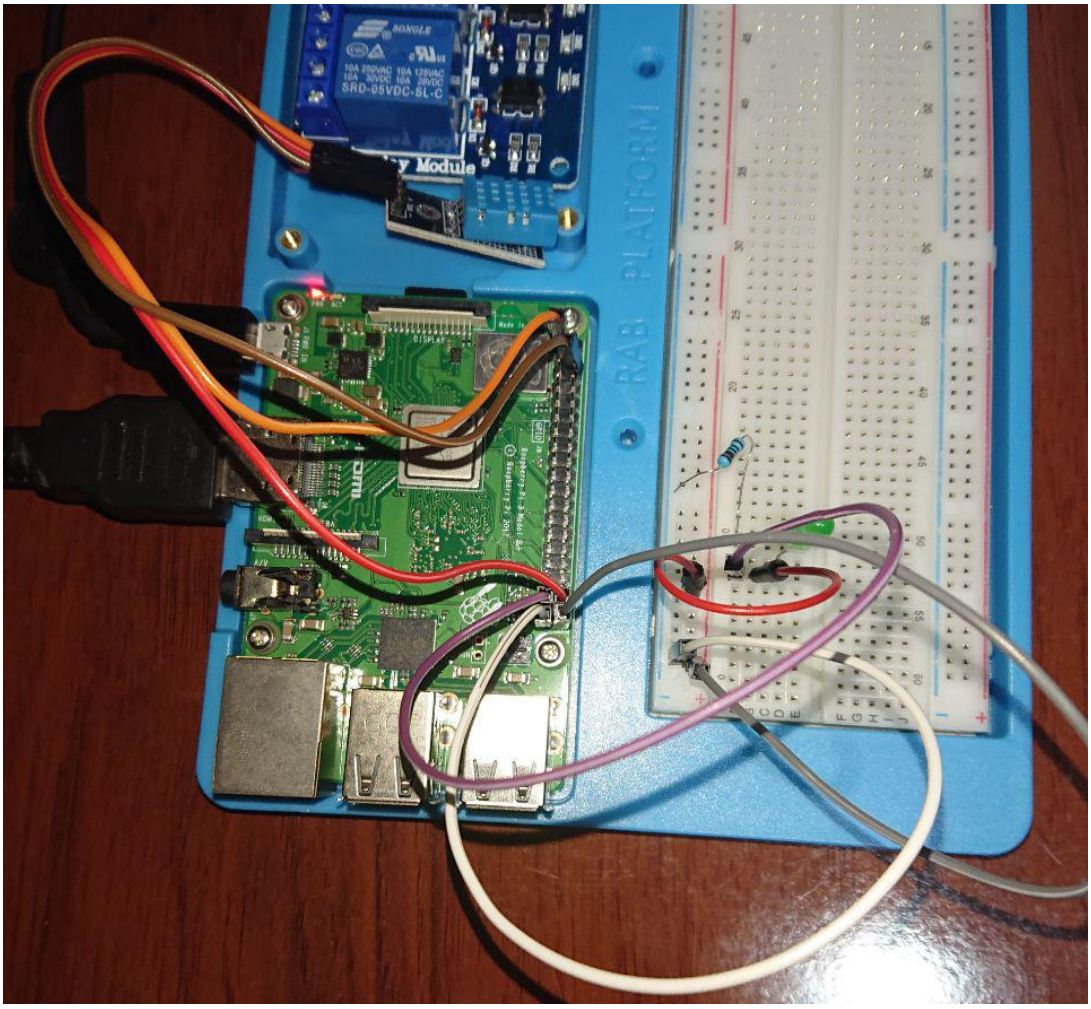


Figura 75: Conexionado Raspberry Pi

ANEXO B: INSTALACIÓN Y CONFIGURACIÓN DE ECLIPSE KURA EN RASPBERRY PI

En este anexo se explica paso a paso cómo instalar y configurar Eclipse Kura 4.1.0 en una Raspberry Pi 3, así como los parámetros necesarios para su configuración.

7.1 Instalación de Eclipse Kura 4.1.0 en una Raspberry Pi 3 con Raspbian Buster

Se instalará Eclipse Kura en una Raspberry Pi 3 con la imagen Raspbian Buster, versión Julio de 2019 [36] para convertir este dispositivo en una pasarela IoT. Para ello se han seguido los pasos indicados en la documentación de Eclipse Kura. [37]

Desde una terminal en la Raspberry Pi ejecutamos los siguientes comandos:

1. Desinstalamos paquetes que son incompatibles con Eclipse Kura.

```
$ sudo apt-get purge dhcpcd5
```

2. Instalamos la herramienta gdebi, necesaria para el funcionamiento.

```
$ sudo apt-get update
$ sudo apt-get install gdebi-core
```

3. Instalamos Java 8 en caso de no tenerlo instalado.

```
$ sudo apt-get install openjdk-8-jre-headless
```

4. Descargamos el paquete de Eclipse Kura y lo instalamos usando la herramienta gdebi

```
$ wget http://download.eclipse.org/kura/releases/<version>/kura_4
.1.0_raspberry-pi-2-3_installer.deb
$ sudo gdebi kura_4.1.0_raspberry-pi-2-3_installer.deb
```

Tras ejecutar estos comandos tendríamos instalado Eclipse Kura en nuestra Raspberry Pi. A continuación, se puede reiniciar y acceder a la interfaz Web de Kura haciendo uso de un navegador Web, haciendo uso de una de las direcciones IP que tenga asignada, o en nuestro caso, la local. Se debe mostrar el contenido de la Figura 76: Interfaz inicio sesión de Eclipse Kura.

```
https://localhost
```

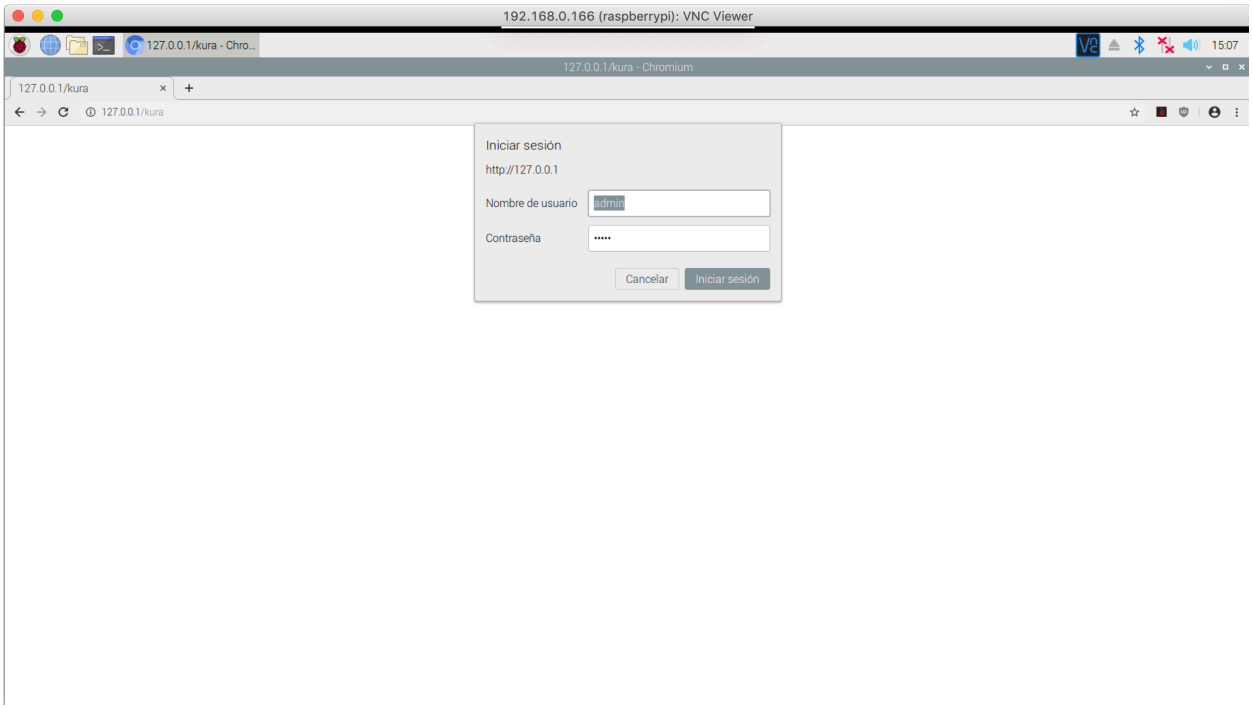


Figura 76: Interfaz inicio sesión de Eclipse Kura

Se puede acceder a la interfaz web haciendo uso del usuario `admin` y contraseña `admin` por defecto. Tras esto, se debe mostrar la Figura 77: Interfaz web de Eclipse Kura.

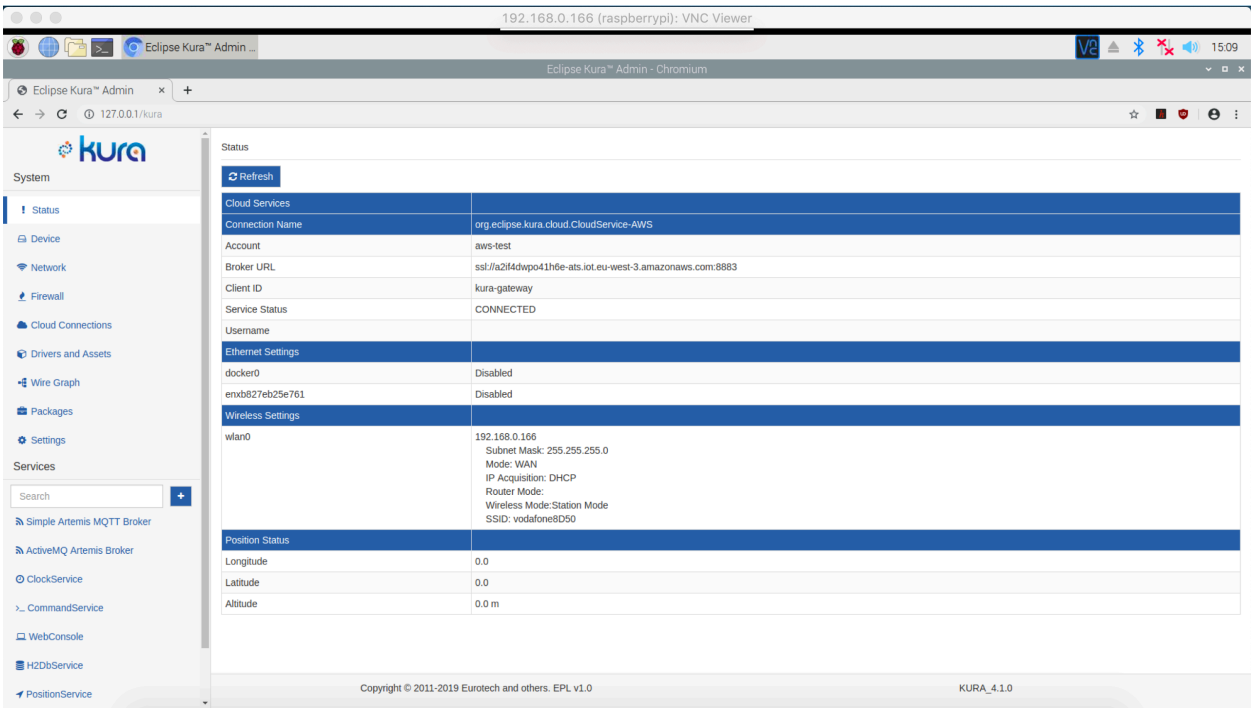


Figura 77: Interfaz web de Eclipse Kura

7.2 Configuración de Eclipse Kura 4.1.0 mediante interfaz Web.

La configuración de red, los bundles implementados [26], las conexiones con servicios en la nube o los registros de errores, entre otros, pueden ser configurados mediante línea de comandos en la RaspberryPi,

o mediante la interfaz web que despliega. Por su facilidad de uso, se configurarán mediante la interfaz web.

7.2.1 Configuración de red

La configuración de red de la Raspberry Pi al instalar Eclipse Kura no se puede hacer mediante los ficheros de configuración como en cualquier sistema operativo Linux ya que esta entra en conflicto con estos [38]. Para ello se hace la configuración a través de la interfaz web de Eclipse Kura.

En el apartado Network del menú lateral izquierdo podremos configurar las interfaces que tenga nuestro dispositivo (ethernet, wlan, dockers, …).

En nuestro caso conectaremos por la RaspberryPi por Wi-Fi con un punto de acceso a Internet. Para configurar las interfaces se ha seguido la configuración indicada en [38], en el caso de una conexión que utiliza el servidor DHCP del router para obtener la dirección IP. Se ve reflejado en las figuras: Figura 78: Configuración Wi-Fi eligiendo punto de acceso y Figura 79: Parámetros configuración interfaz Wi-Fi:

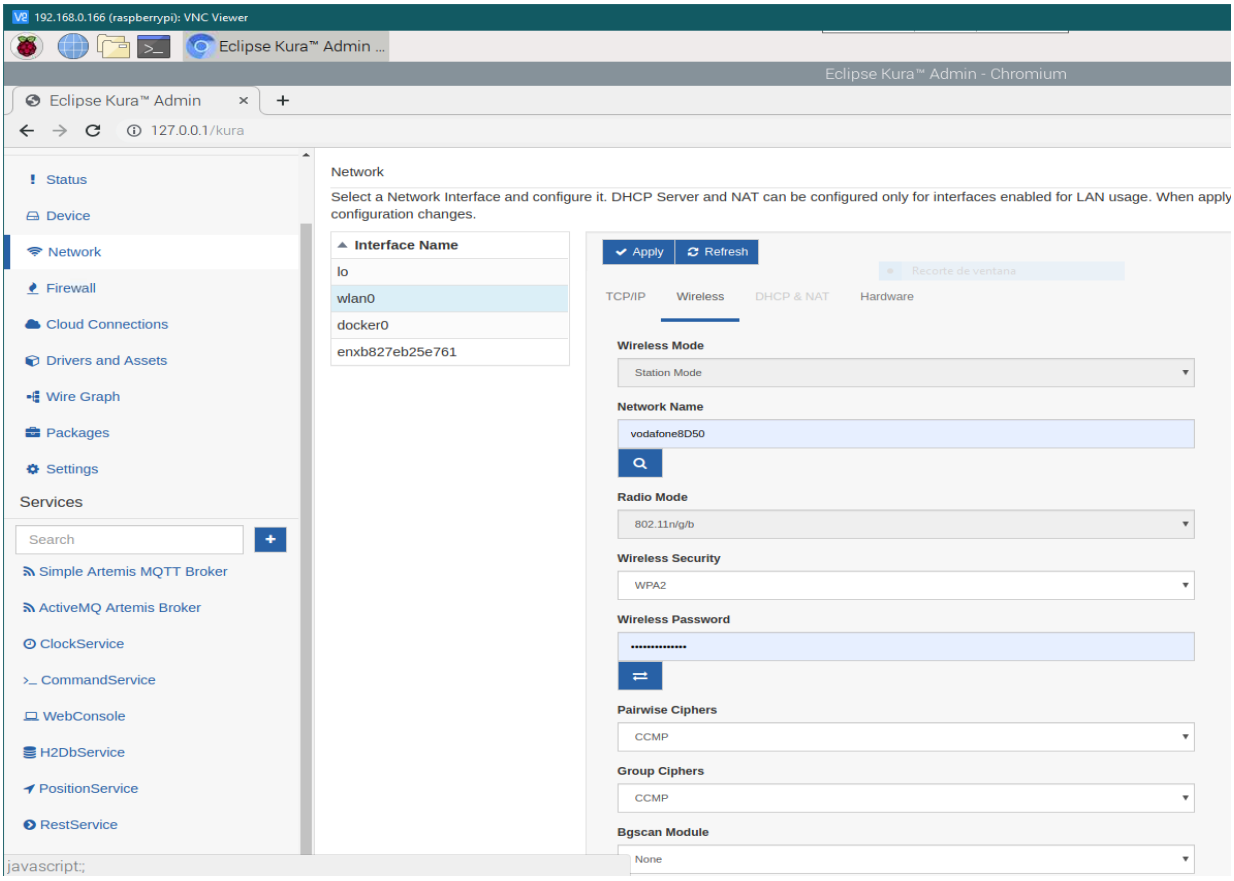


Figura 78: Configuración Wi-Fi eligiendo punto de acceso

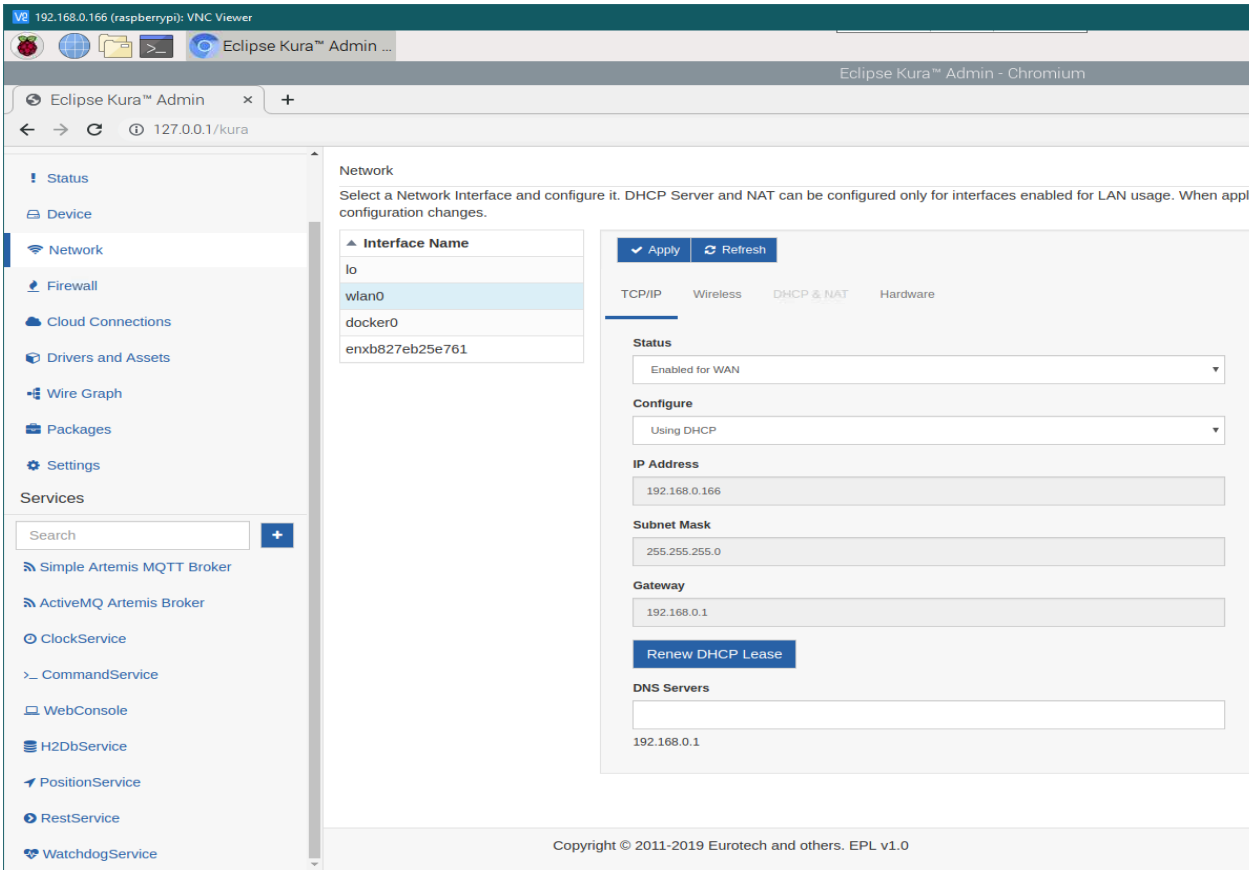


Figura 79: Parámetros configuración interfaz Wi-Fi

7.2.2 Configuración para la compatibilidad con AWS

Para que podamos hacer uso del módulo de Cloud Connections que incorpora Eclipse Kura. Este proporciona un servicio para conectarnos de forma sencilla con una nube, en nuestro caso, Amazon AWS.

Gracias a este módulo podemos hacer uso de los bundles que implementemos en la nube. Para ello se debe haber configurado antes, haciendo uso de los certificados de seguridad, indicar cual es el punto final de conexión, entre otros. En el Anexo C: Creación de usuarios y configuración de servicios en AWS se indica como se obtienen estos.

Para hacer estas configuraciones tenemos que acceder al apartado Cloud Connections del menú lateral izquierdo. Lo configuraremos según se indica en [39].

Los pasos a seguir para que nuestro Dispositivo Kura se conecte con el endpoint de nuestra nube son:

1. Crear un almacén de claves Java (keystore) e incluir el certificado raíz.

Para crear el almacén de claves, tenemos que descargar el certificado raíz de Amazon AWS IoT, se puede realizar de manera sencilla ejecutando el siguiente comando:

```
$ curl https://www.amazontrust.com/repository/AmazonRootCA3.pem > /tmp/root-CA.pem
```

A continuación, procedemos a generar el almacén de claves con nombre **cacerts** y contraseña **juan18**:

```
$ sudo mkdir /opt/eclipse/kura/security
```

```
$ cd /opt/eclipse/kura/security

$ sudo keytool -import -trustcacerts -alias aws -file /tmp/root-CA.pem -keyst
ore cacerts -storepass juan18
```

Como resultado, podemos observar en la Figura 80: Almacén de claves Raspberry Pi que nuestro almacén de claves ha sido creado y contiene el certificado raíz.

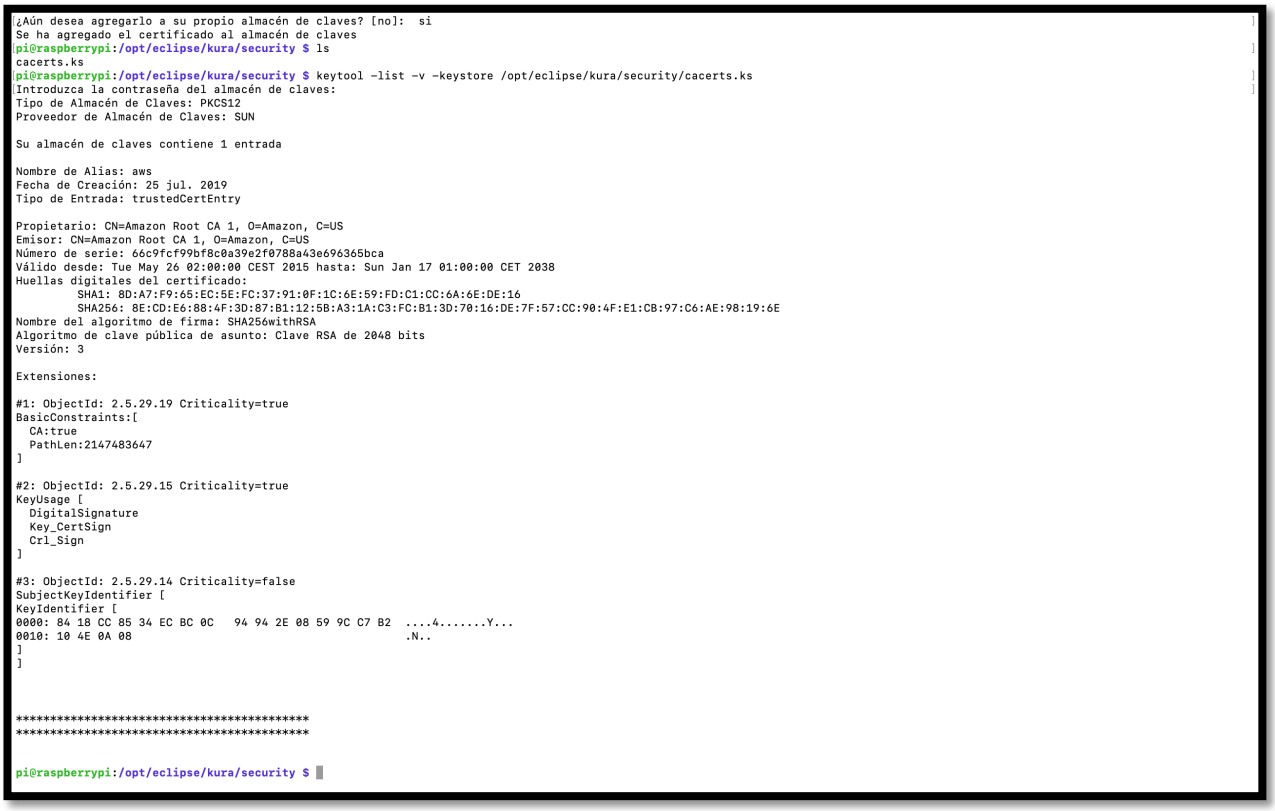


Figura 80: Almacén de claves Raspberry Pi

2. Configurar el apartado SSL y la identidad de nuestro Dispositivo Kura. Para ello desde el menú izquierdo de la interfaz Web de Eclipse Kura debemos ir a: Settings -> SSL Configuration.

En este indicamos:

- ssl.default.truststore: /opt/eclipse/kura/cacerts.ks
- ssl.keystore.password: juan18

En el desplegable que aparece, debemos ir a Device SSL Certificate. En este introducimos un alias, en este caso `aws-ssl`, el certificado (contenido de `1a0bee08ca-certificate.pem.crt`) y la clave privada que se nos ha proporcionado al crear un objeto en Amazon AWS, en formato PKCS8. Para ello ejecutamos el comando:

```
$ openssl pkcs8 -topk8 -inform PEM -outform PEM -in 1a0bee08ca-pr
ivate.pem.key -out outKey.pem -nocrypt
```

3. En el apartado Cloud Connections del panel izquierdo configuramos una nueva conexión a la nube.

En este apartado existe una lista de los servicios cloud implementados. Como ya dijimos, Eclipse Kura implementa uno y nos permite usarlo desde nuestros bundles.

Vemos que existen tres pestañas: CloudService, DataService y MqttDataTransport.
Haciendo click en Nueva Conexión e insertamos los parámetros que se muestran en la Figura 81: Nuevo Cloud Connection.

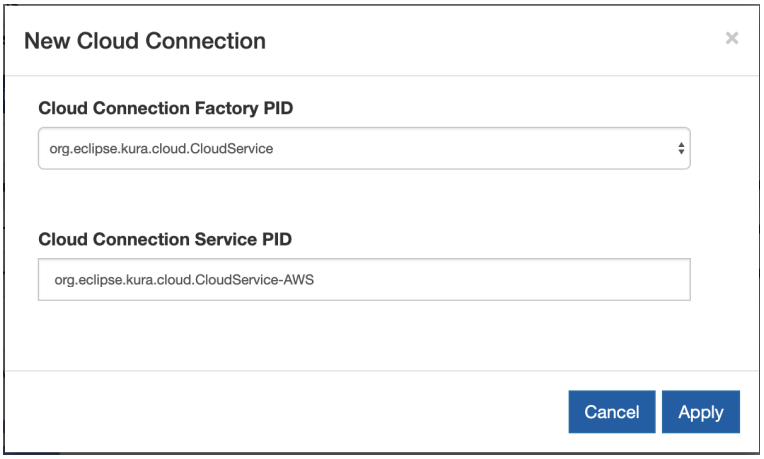


Figura 81: Nuevo Cloud Connection

Como la información la vamos a enviar mediante MQTT configuramos la pestaña MQTTDataTransportAWS:

Los parámetros por introducir son:

- Broker-url: Punto de enlace personalizado. Se puede obtener en el apartado Configuración del IoT Core de AWS. En nuestro caso, la dirección que se muestra en la Figura 82: Punto de enlace personalizado.

```
mqtts://a2if4dwpo41h6e-ats.iot.eu-west-3.amazonaws.com:8883/
```

Figura 82: Punto de enlace personalizado

- Topic Context Account-Name: Puede suprimirse. En nuestro caso: `aws`.
- Client-Id: Lo utilizaremos para establecer la ruta en la que publicar/suscribirnos: `things`.
- SSL Certificate Alias: Alias introducido en la configuración del certificado SSL: `aws-ssl`.

Con esto, nuestro sistema estaría listo para conectarse a los Cloud Services de Amazon AWS.

ANEXO C: CREACIÓN DE USUARIOS Y CONFIGURACIÓN DE SERVICIOS EN AWS

En este anexo se explica paso a paso cómo crear un usuario y crear y configurar los distintos servicios utilizados en Amazon AWS. Para el desarrollo de este Trabajo de Fin de Grado, todos los servicios utilizados han sido gratuitos, ya que no se ha excedido el límite de recursos gratuitos que la plataforma proporciona.

7.3 Creación de usuario AWS

Para crearnos un usuario en la plataforma de Amazon AWS, tenemos que dirigirnos a la siguiente URL: https://aws.amazon.com/free. Una vez en esta, podemos ver todos los productos que AWS oferta en su plataforma de Cloud Computing. Para crear una cuenta, debemos hacer click en el botón naranja de la esquina superior derecha (Crear una cuenta de AWS).

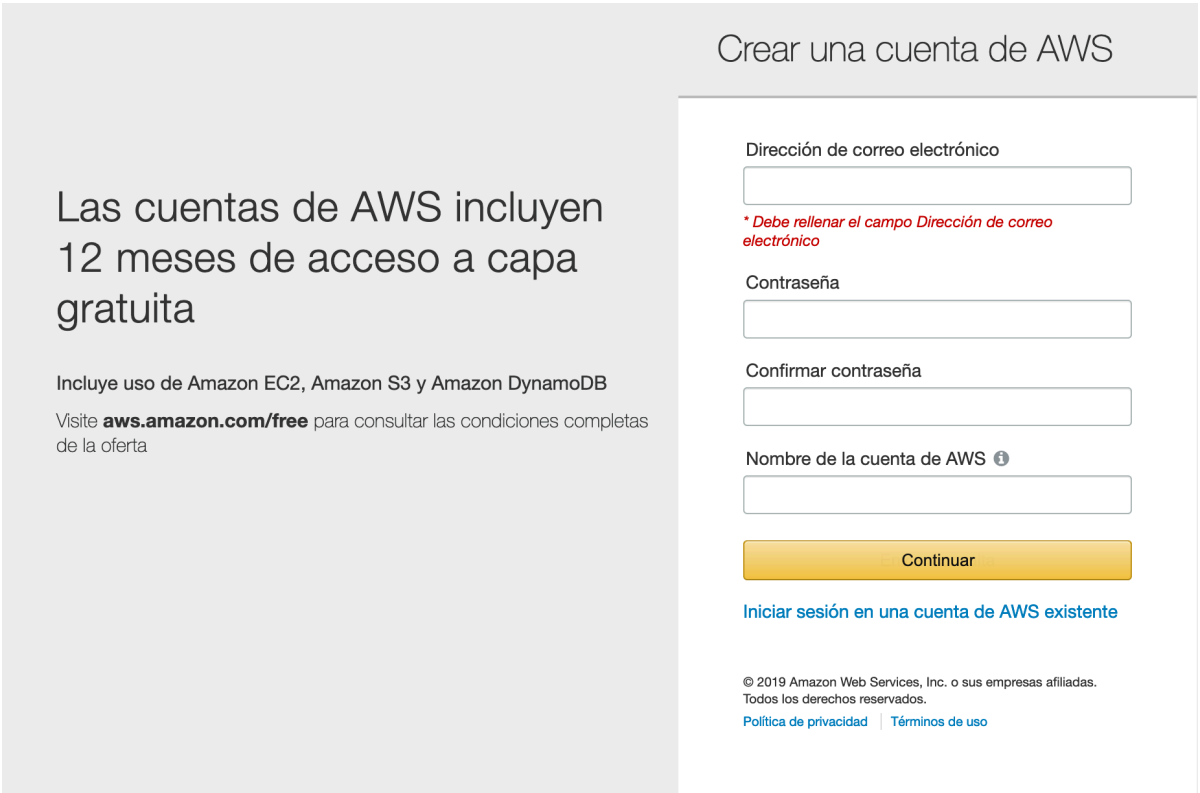


Figura 83: Formulario AWS

Se deberá rellenar el formulario que aparece en la Figura 83: Formulario AWS con la información personal del usuario a crear. Se deberá rellenar toda la información que se solicite en cada paso. Pide una tarjeta de crédito para verificar la identidad, pero no se hará ningún cargo a esta. Una vez hayamos completado el registro, tendremos nuestra sesión creada. Se nos mostrará nuestro panel principal, como podemos ver en la Figura 84: Panel principal AWS.

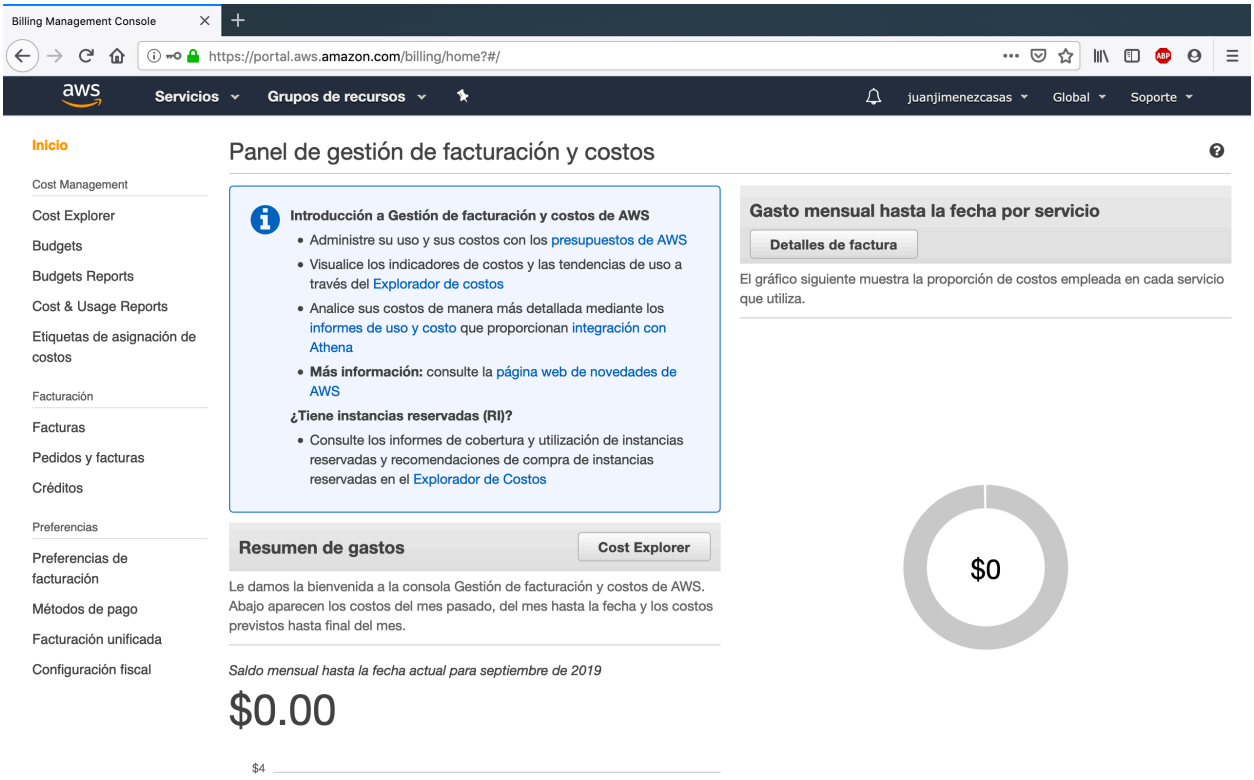


Figura 84: Panel principal AWS

7.4 IoT Core

En este apartado se indican los pasos necesarios para crear un Objeto en el IoT Core de AWS. Este objeto será el utilizado en el entorno de Cloud Computing de AWS para representar un dispositivo IoT, en este caso, la Raspberry Pi con Eclipse Kura.

Para empezar a utilizar IoT Core, debemos hacer click en el elemento Servicios que aparece en la Figura 84: Panel principal AWS. Ahí introduciremos en el buscador IoT Core. Se nos debe mostrar lo que aparece en la Figura 85: IoT Core.

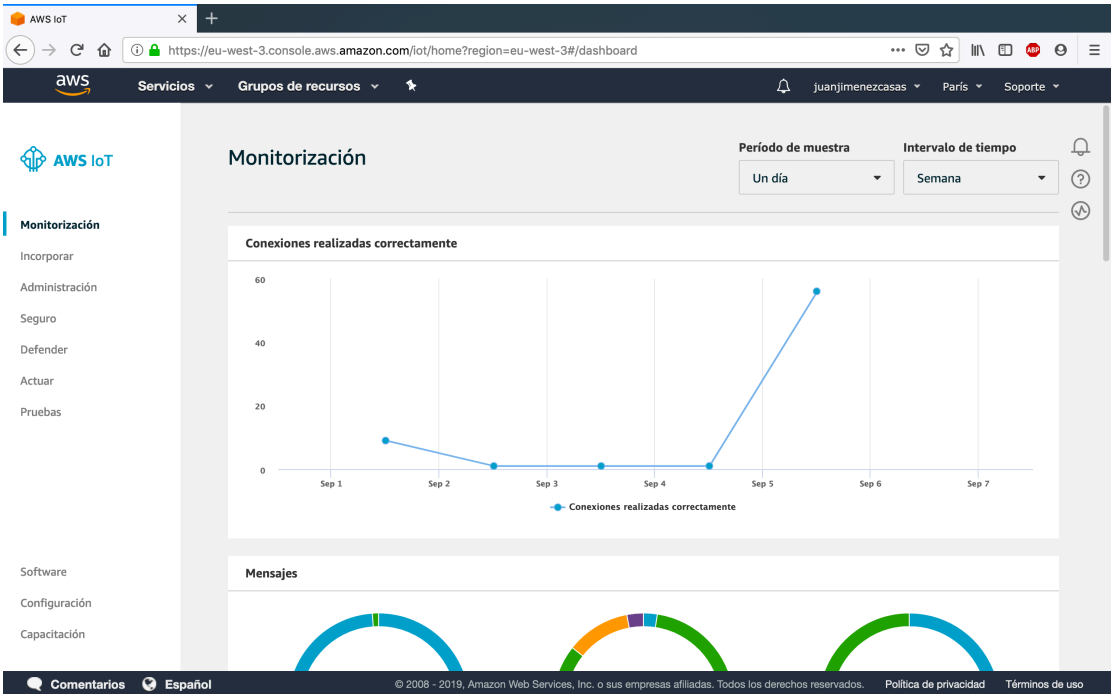


Figura 85: IoT Core

Una vez en este, debemos crear un Objeto. Para ello debemos ir a Administración -> Objetos -> Crear. Debemos elegir la opción crear un solo Objeto. Habrá que rellenar cada uno de los campos que se solicitan en la Figura 86: Creación Objeto.

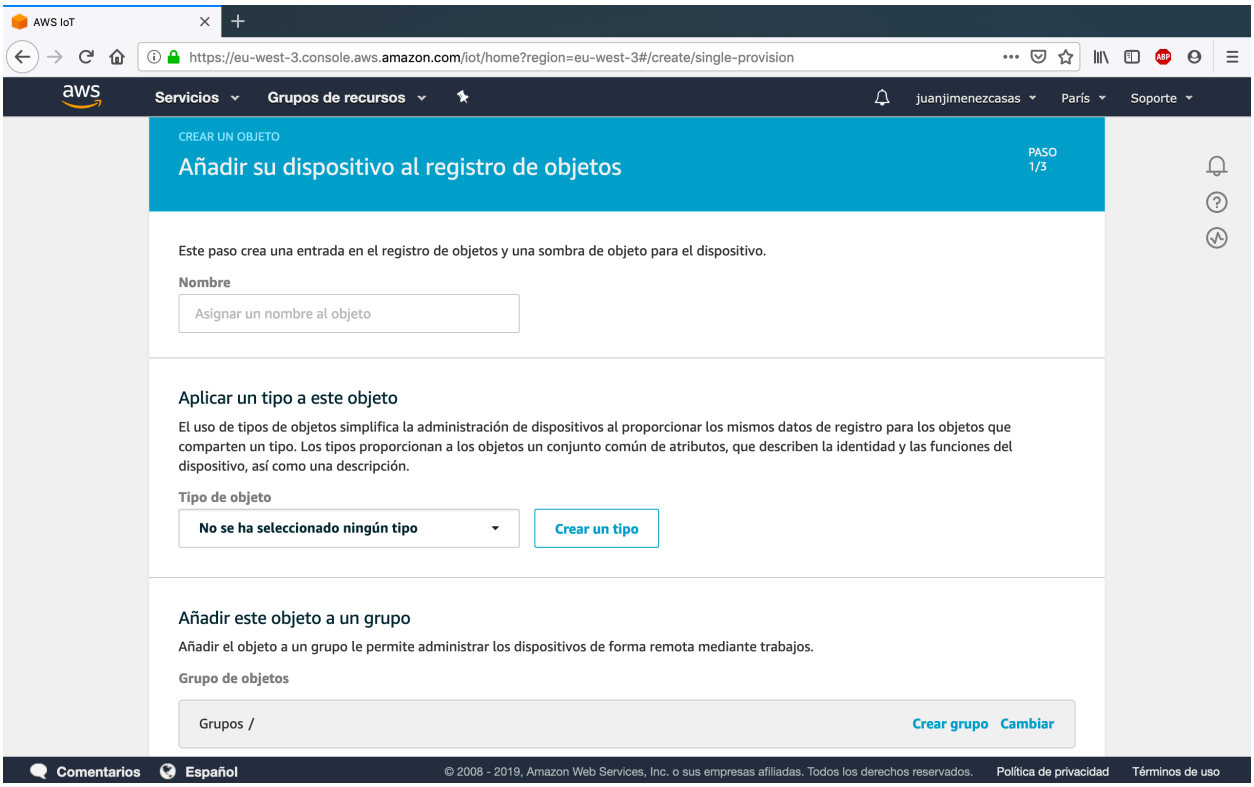


Figura 86: Creación Objeto

En este caso, le asignaremos el nombre **kura-gateway**. Este objeto representará la Raspberry Pi, ya que es un Gateway IoT, haciendo uso de Eclipse Kura. No le aplicaremos ningún tipo. No es necesario asignarle ahora un grupo al que pertenecer. A continuación, debemos darle a Siguiente.

Para que este Objeto adquiera una Identidad, que nos permita que se conecten a este dispositivo IoT de forma segura, debemos asignarle unos certificados. La forma más fácil es hacerlo a través de la opción que proporciona Amazon AWS. Para ello debemos hacer click en **Crear certificado en un click**. Debemos de **descargarnos** y **activar** los certificados que aparecen en la Figura 28: Certificado asociado a un objeto.

Estos son los que utilizaremos en el Anexo B: Instalación y configuración de Eclipse Kura en Raspberry Pi, apartado 7.2.2 y en el apartado 5.1.1 Interacción con el Objeto: Thing Shadow.

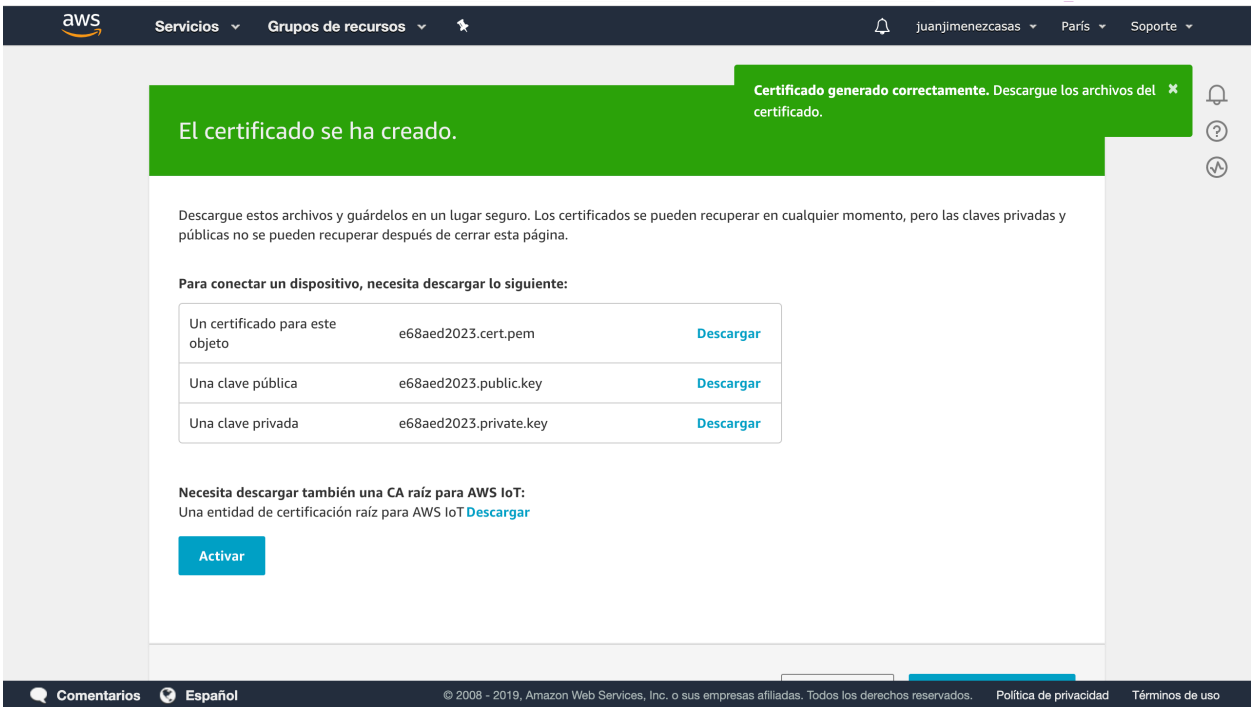


Figura 28: Certificado asociado a un objeto

Por último, debemos asociarle unas políticas al objeto. Las políticas asociadas son las que se muestran en la Figura 29: Política asociada a un certificado. Estas son las políticas por defecto que se incluyen en la documentación de AWS [30].

Con todo esto, ya tendríamos nuestro Objeto **kura-gateway** configurado y con las direcciones que se muestran en la Figura 31: Servicio Device Shadow, para que se pueda interactuar con estos. La dirección que aparece en el apartado HTTPS coincide con la dirección de **punto de enlace personalizado**.

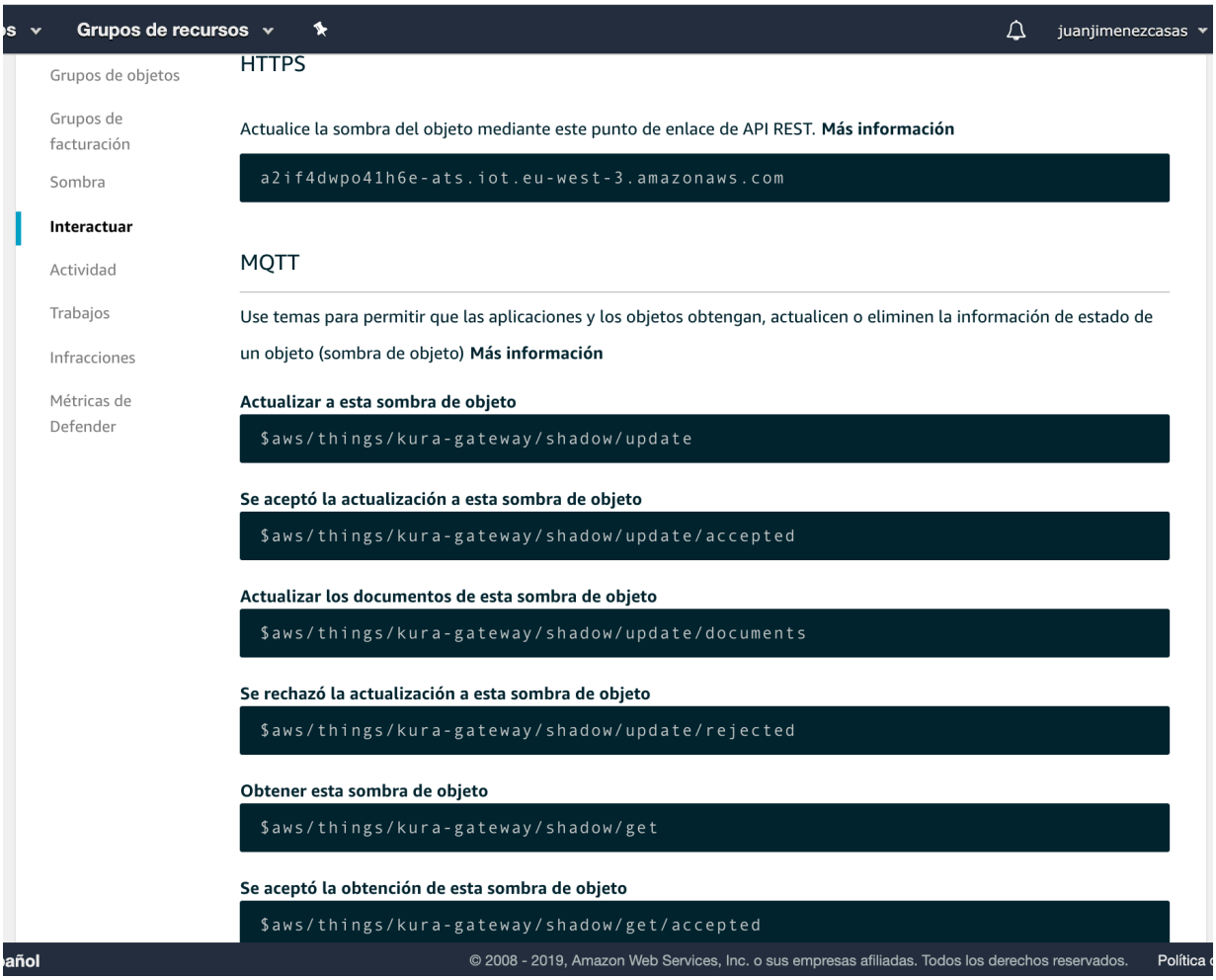


Figura 31: Servicio Device Shadow

7.5 DynamoDB

En este apartado se indica cómo crear una base de datos NoSQL en AWS. Utilizaremos la Base de Datos gratuita que se proporciona en este entorno. En ella almacenaremos la información relativa a un terrario.

Para empezar a utilizar DynamoDB debemos hacer click en el elemento Servicios que aparece en la Figura 84: Panel principal AWS. Ahí introduciremos en el buscador DynamoDB. Se nos debe mostrar lo que aparece en la Figura 87: Panel de control DynamoDB.

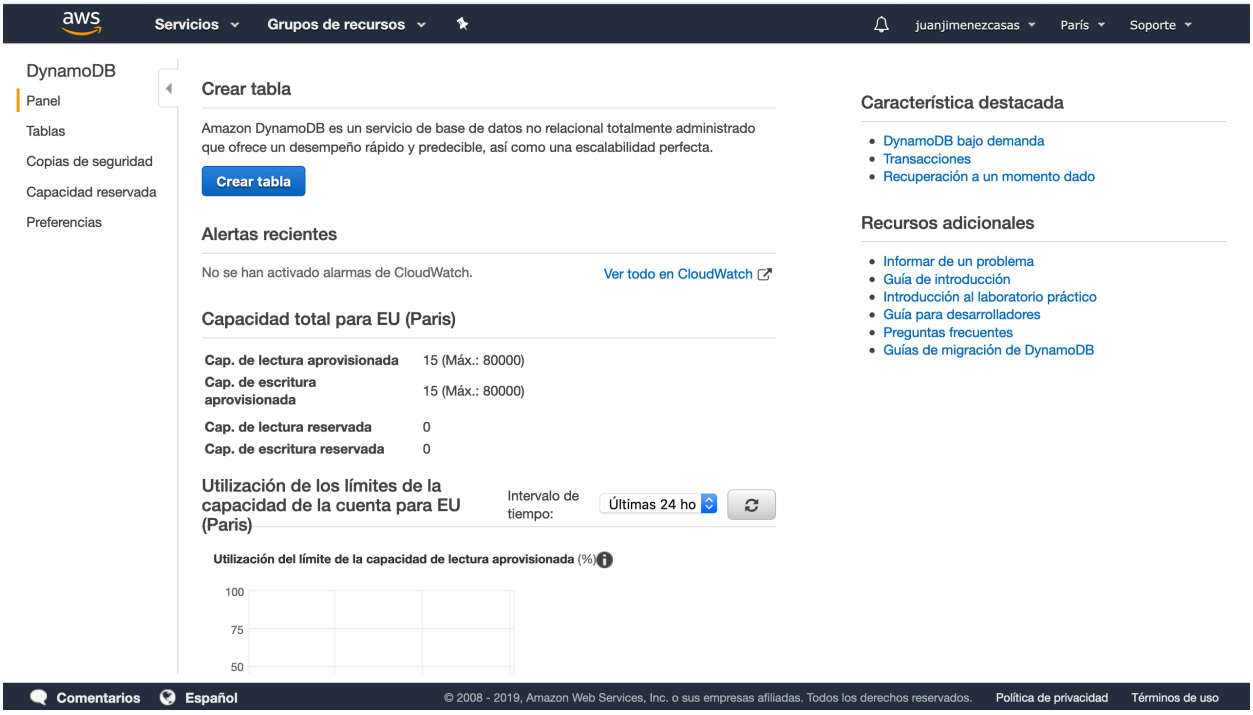


Figura 87: Panel de control DynamoDB

El primer paso es crear una tabla. Para ello hacemos click en el botón que indica **Crear tabla**. Debemos rellenar los campos tal y como se muestran en la Figura 88: Parámetros tabla terrario. Con estos parámetros, la tabla se corresponderá con la indicada en el apartado 5.2 Dynamo DB.

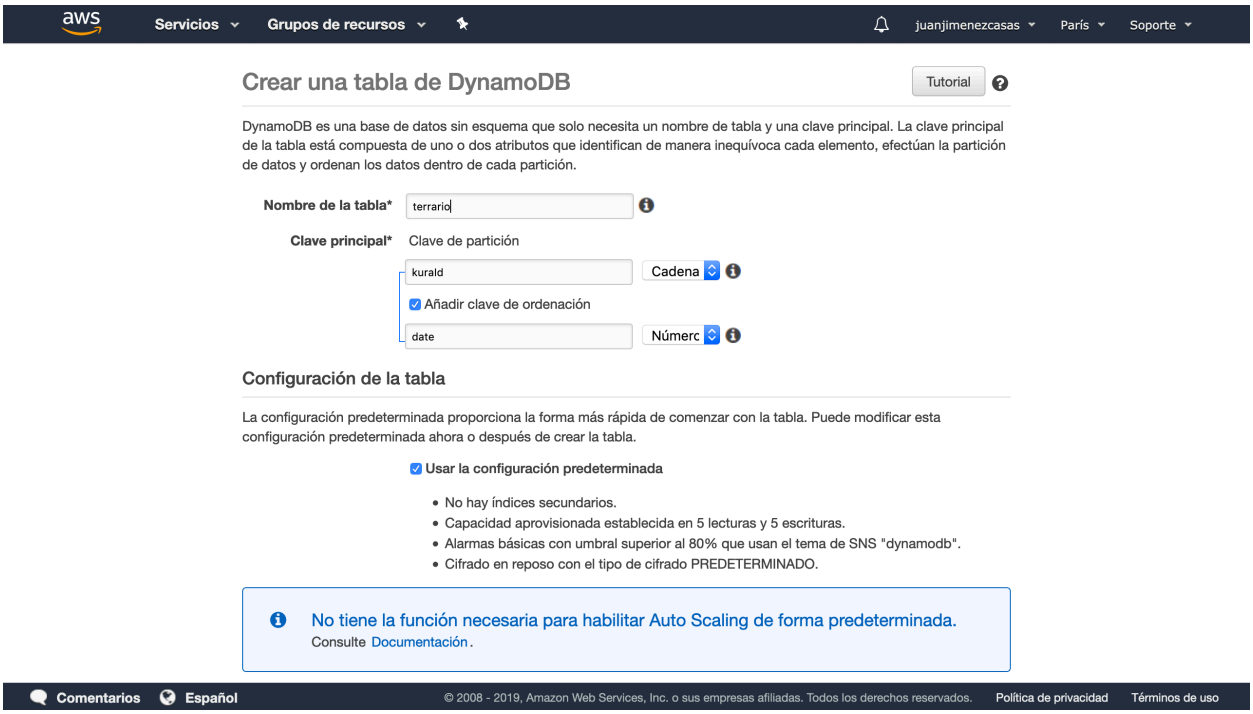


Figura 88: Parámetros tabla terrario

Una vez creada, nos debe aparecer la información de esta tal y como se muestra en la Figura 34: Información Base de Datos terrario. Como podemos apreciar, se nos proporciona un **Nombre de recurso de Amazon (ARN)**. Este es el que utilizaremos para referirnos a esta tabla, desde cualquier punto del entorno de Amazon AWS (asociar políticas o acceder desde las funciones Lambda).

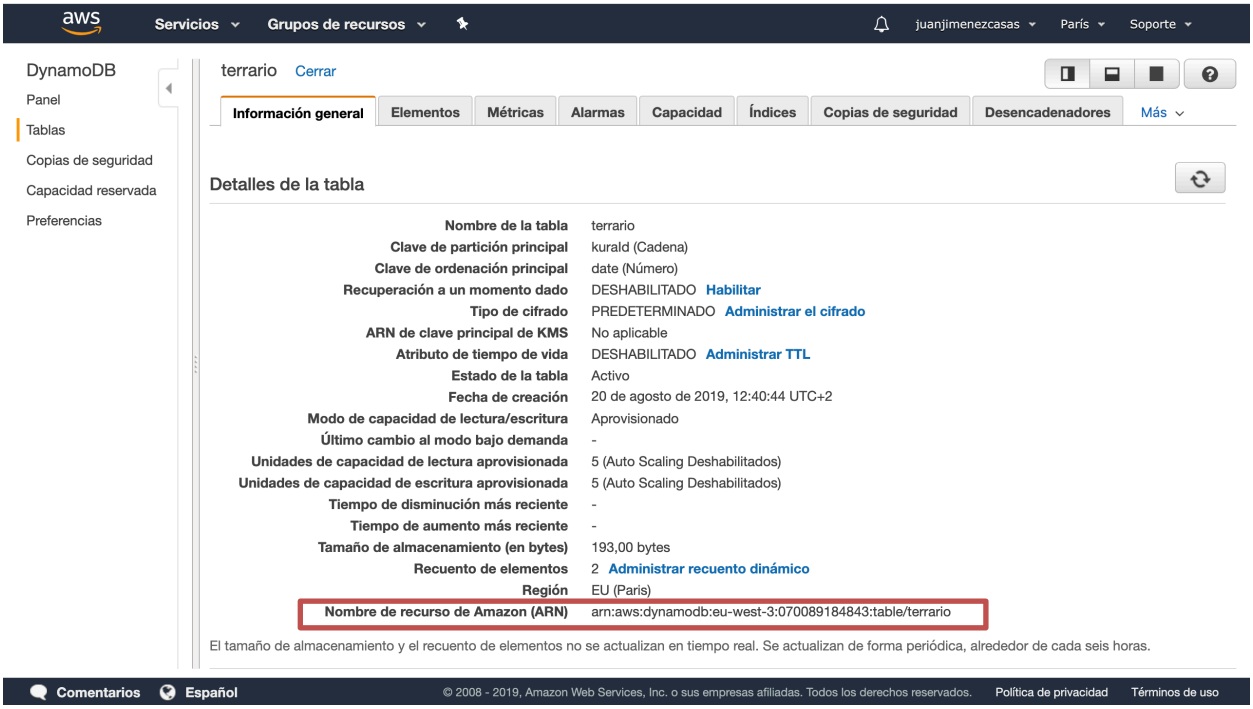


Figura 34: Información Base de Datos terrario

7.6 Funciones Lambda

En este apartado se indica cómo crear una función Lambda en AWS. Escogeremos como lenguaje de programación Node.js. Además, se verá como asignarle los permisos necesarios para que pueda acceder a los recursos que necesite (IoT Core o DynamoDB).

Para empezar a utilizar Lambda debemos hacer click en el elemento Servicios que aparece en la Figura 84: Panel principal AWS. Ahí introduciremos en el buscador Lambda. Se nos debe mostrar lo que aparece en la Figura 89: Panel de control Lambda.

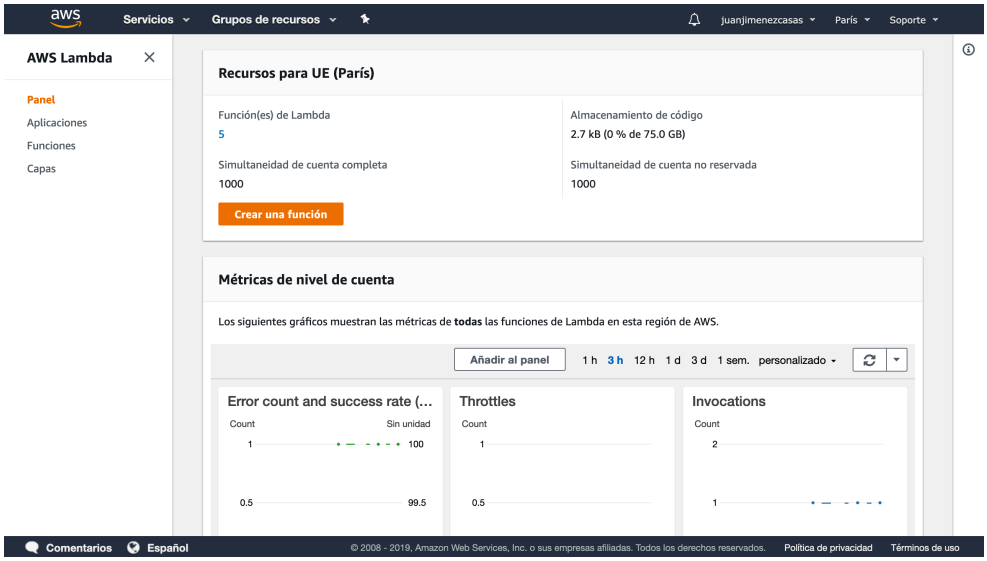


Figura 89: Panel de control Lambda

Para crear una función debemos hacer click en el botón **Crear una función**. Debemos rellenar los campos con la información que se muestra en la Figura 90: Crear función. En el campo de **Rol**, para la

primera función se deberá crear uno nuevo. Para el resto de las funciones se puede utilizar el mismo si se quieren compartir los permisos que se le asignen a ese Rol.

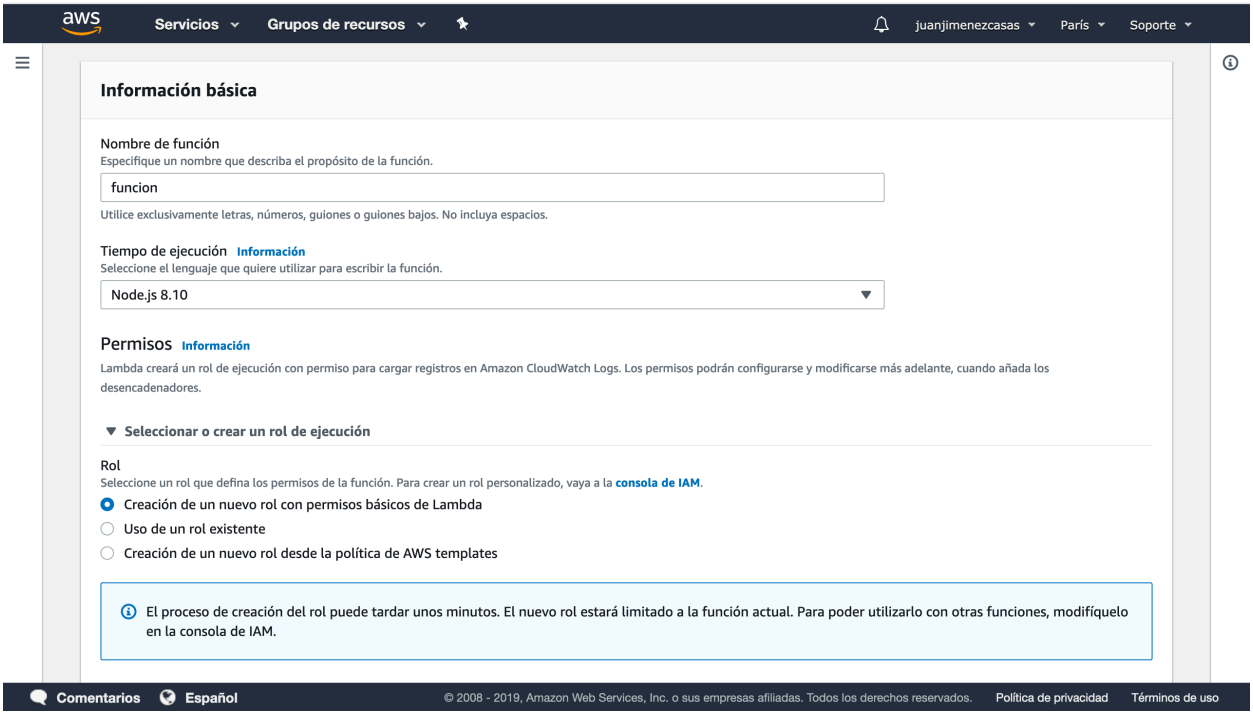


Figura 90: Crear función

Una vez se haya creado la función, deberá aparecernos un panel como se muestra en la Figura 91: Panel de control de una función Lambda. Lo primero que debemos hacer es asignarle los desencadenadores que pueda tener la función (API Gateway, AWS IoT o DynamoDB). Ahí le asignaremos la regla o la petición que la invoque.

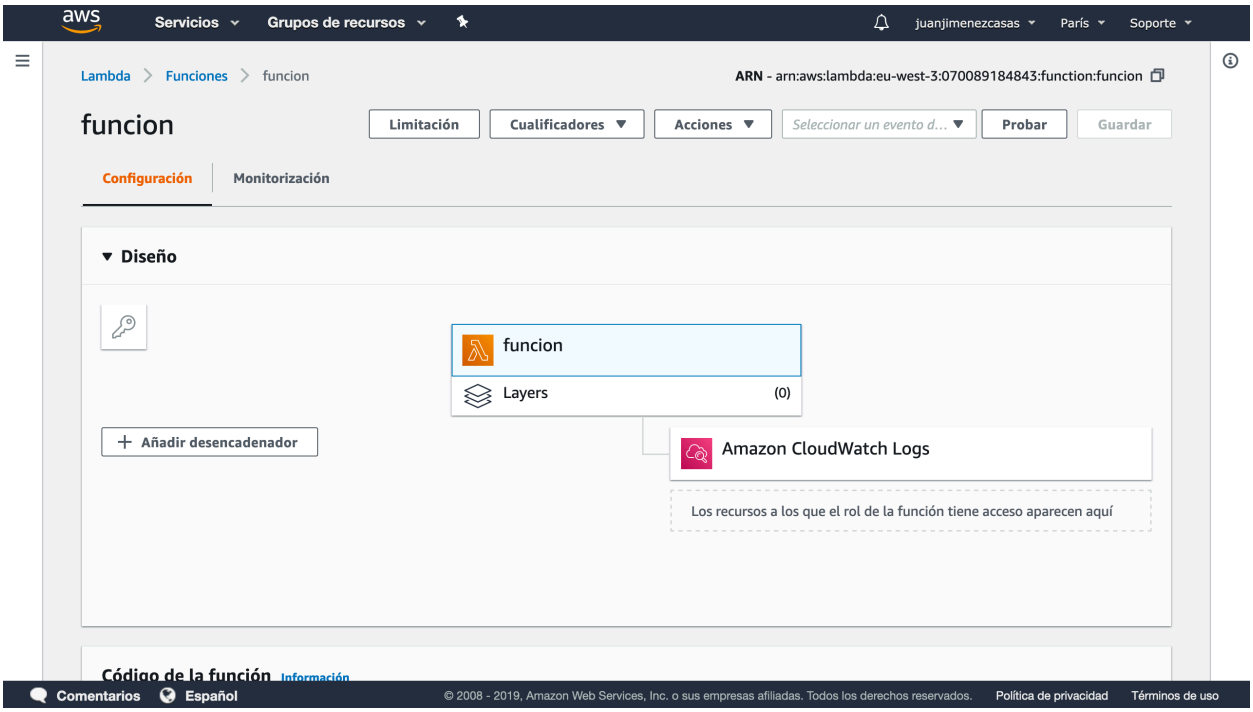


Figura 91: Panel de control de una función Lambda

Posteriormente, en el editor de texto que proporciona, implementaremos lo que queremos que la función haga en cada caso. Para que todo funcione correctamente, al Rol de la función tenemos que darle

permisos para que acceda a los recursos a los que se necesite acceder en cada caso. Esto se hace tal y como se indica en el tutorial de Amazon AWS para crear una función Lambda [40]. Para ello, tenemos que ir al apartado que se muestra en la Figura 92: Rol de ejecución Lambda. Haciendo click en **Consulte el rol (nombre_rol)**, nos reenviará al servicio que nos permite gestionarlo, que es AWS IAM.

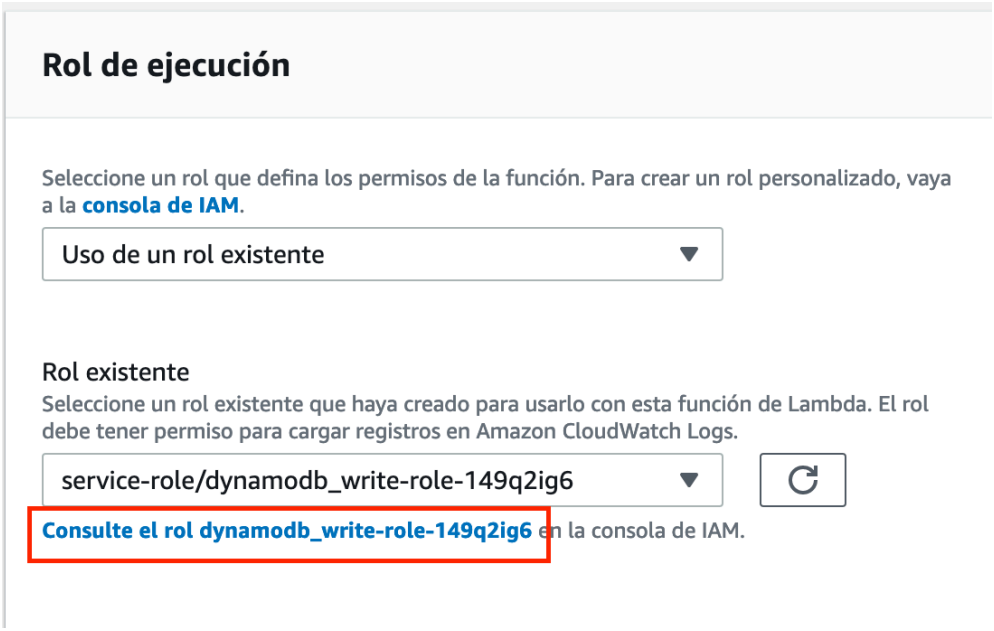


Figura 92: Rol de ejecución Lambda

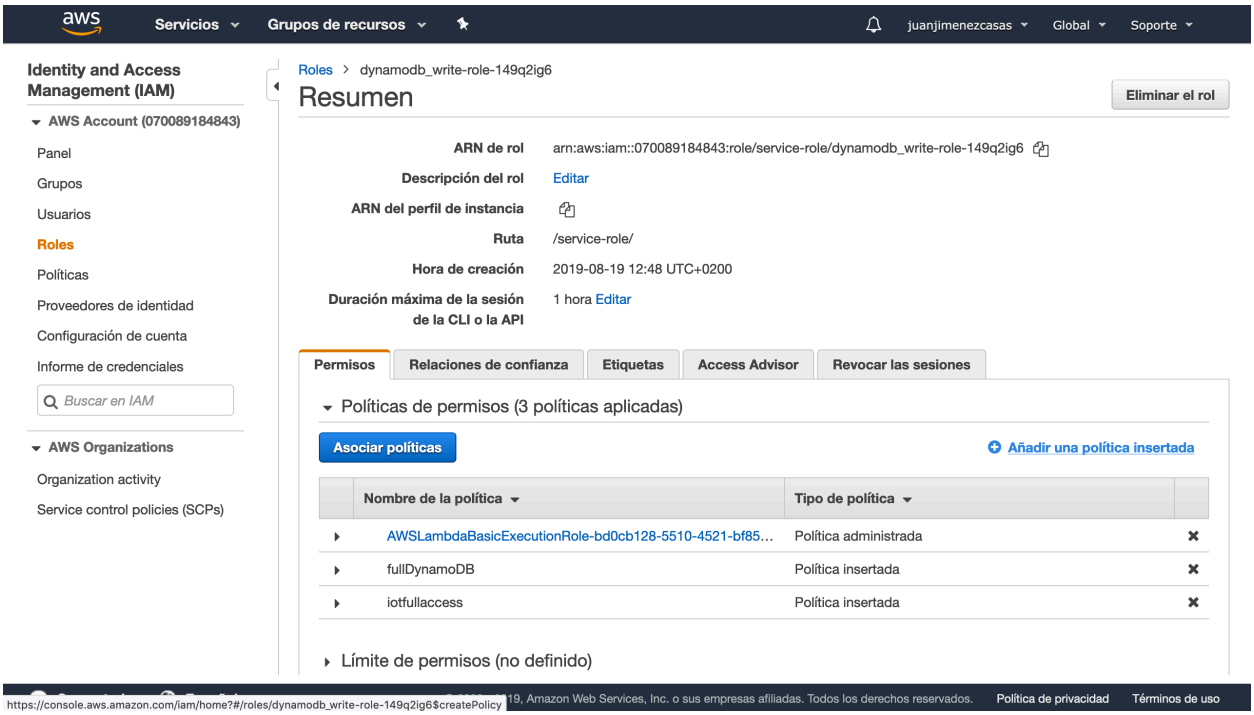


Figura 93: Resumen Rol

En este se nos mostrará un panel como el que se muestra en la Figura 93: Resumen Rol. Como podemos apreciar, tiene 3 políticas insertadas.

- AWSLambdaBasicExecutionRole-bd0cb128-5510-4521-bf85-bf76b8773022: Es la que se le asocia automáticamente al crear el rol con la función Lambda. Esta le asocia los permisos necesarios para que pueda ejecutar la función cuando recibe un evento desencadenador.

- fullDynamoDB: Política creada para que este rol tenga tanto acceso de escritura como de lectura a la Base de Datos terrario. El contenido de esta política es el que se puede ver en la Figura 94: Política fullDynamoDB.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": [
                "dynamodb:*"
            ],
            "Resource": [
                "arn:aws:dynamodb:eu-west-3:070089184843:table/terrario"
            ]
        }
    ]
}
```

Figura 94: Política fullDynamoDB

- iotfullaccess: Política creada para que este rol tenga permisos para publicar/suscribirse a un topic IoT del IoT Core. El contenido de esta política es el que se muestra en la Figura 95: Política iotfullaccess.

```
        {
            "Sid": "VisualEditor1",
            "Effect": "Allow",
            "Action": "iot:*",
            "Resource": [
                "arn:aws:iot:*:*:scheduledaudit/*",
                "arn:aws:iot:*:*:otaupdate/*",
                "arn:aws:iot:*:*:thinggroup/*",
                "arn:aws:iot:*:*:topicfilter/*",
                "arn:aws:iot:*:*:cacert/*",
                "arn:aws:iot:*:*:billinggroup/*",
                "arn:aws:iot:*:*:policy/*",
                "arn:aws:iot:*:*:job/*",
                "arn:aws:iot:*:*:stream/*",
                "arn:aws:iot:*:*:topic/*",
                "arn:aws:iot:*:*:client/*",
                "arn:aws:iot:*:*:thing/*"
            ]
```

Figura 95: Política iotfullaccess

Con estas políticas, las funciones Lambda que tengan asociado este Rol para sus ejecuciones, podrán tener todos los permisos necesarios para acceder a los recursos que se necesitan para este proyecto.

7.7 API Gateway

En este apartado se indica cómo crear una API REST en AWS haciendo uso de API Gateway. Nuestra API REST, contendrá los métodos GET y POST. Además, se verá como asignarle las plantillas de mapeo necesarias para la información contenida en el cuerpo de las peticiones HTTP sea mapeada correctamente a la función Lambda que se encargue de procesar la petición.

Para empezar a utilizar API Gateway, debemos hacer click en el elemento Servicios que aparece en la Figura 84: Panel principal AWS. Ahí introduciremos en el buscador API Gateway. Se nos debe mostrar lo que aparece en la Figura 96: Panel de control API Gateway. Esta API se ha creado siguiendo los pasos de la documentación de AWS [41].

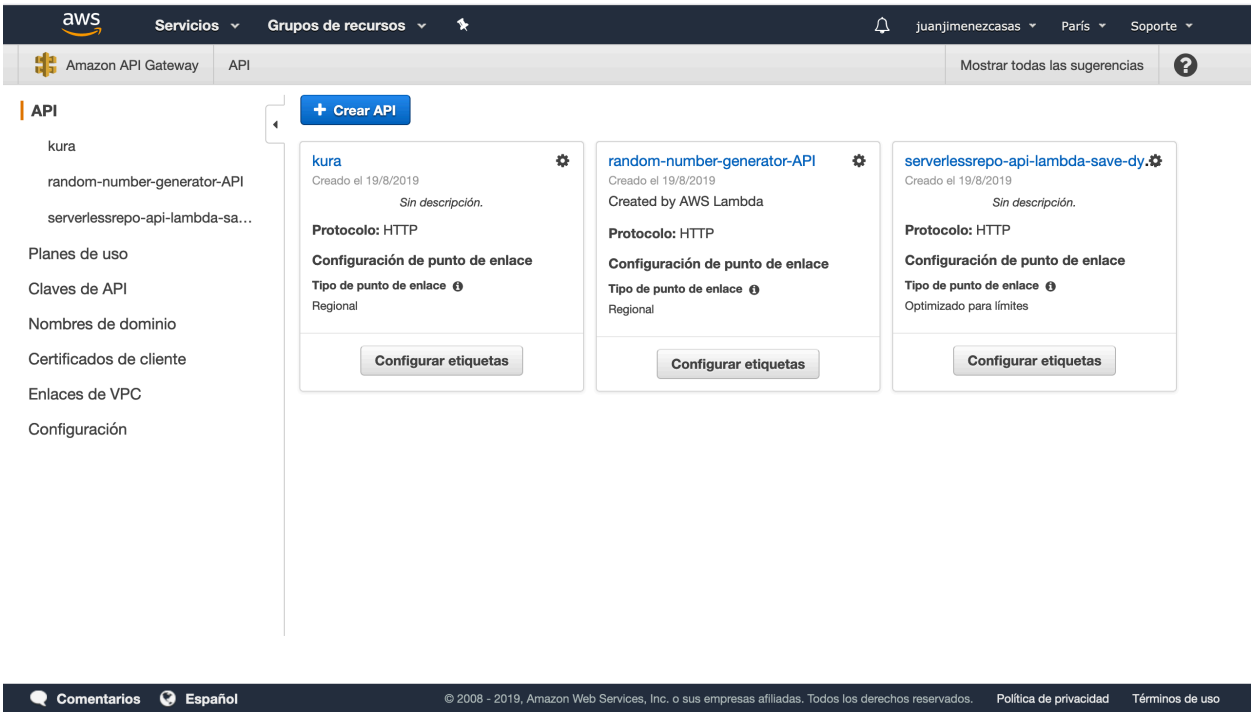


Figura 96: Panel de control API Gateway

Para crear una API REST debemos hacer click en el botón **Crear API.** Debemos rellenar los campos con la información que se muestra en la Figura 97: Creación API REST.

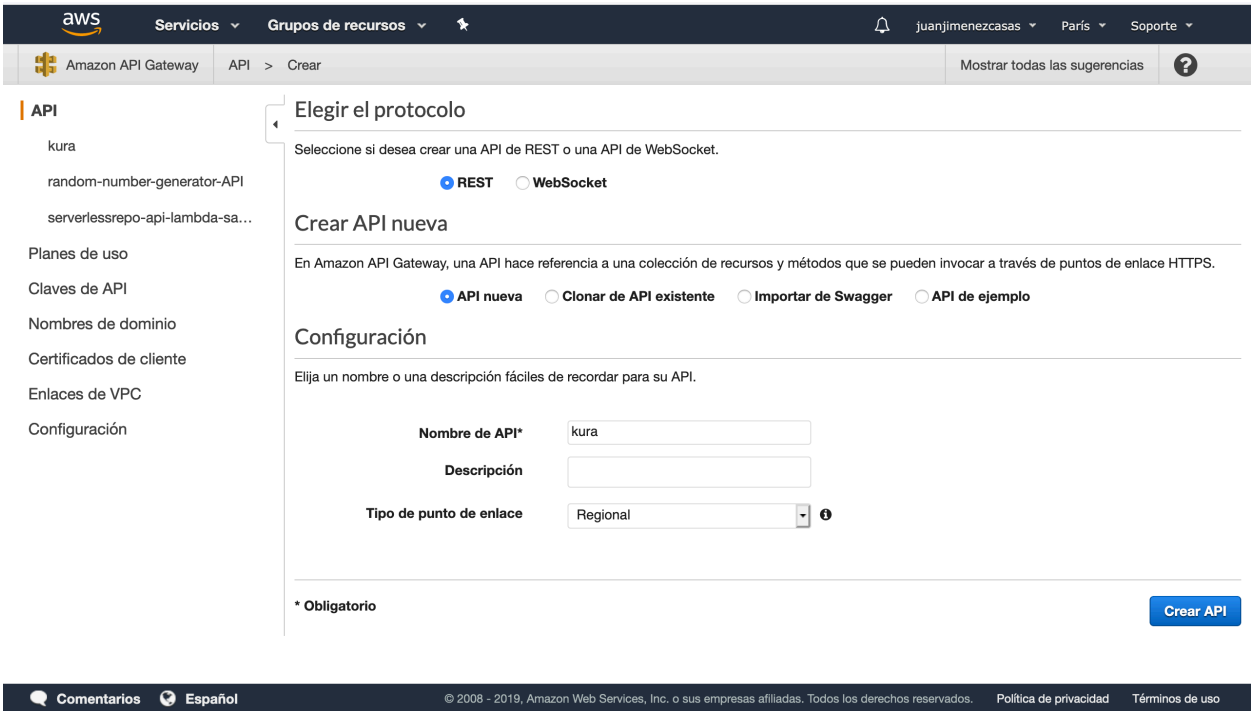


Figura 97: Creación API REST

Una vez creada la API, debemos de indicar los recursos que esta va a ofrecer, así como los métodos para obtenerlos. Para ello, debemos crear primero un recurso. Esto se hace haciendo click en: Acciones->Crear recurso. Para nuestra aplicación lo llamaremos **kura**. Tal y como se indica en la documentación de AWS, se debe marcar la casilla Habilitar API Gateway CORS [41].

Una vez creado el recurso, crearemos los métodos que podrán acceder a este. Esto se hace haciendo click en: Acciones->Crear método y elegimos el método a implementar. Nos mostrará una ventana como la que se muestra en la Figura 98: Creación método API. En esta ventana tenemos que indicar la función Lambda que queremos que se invoque cuando se reciba una petición de ese tipo.

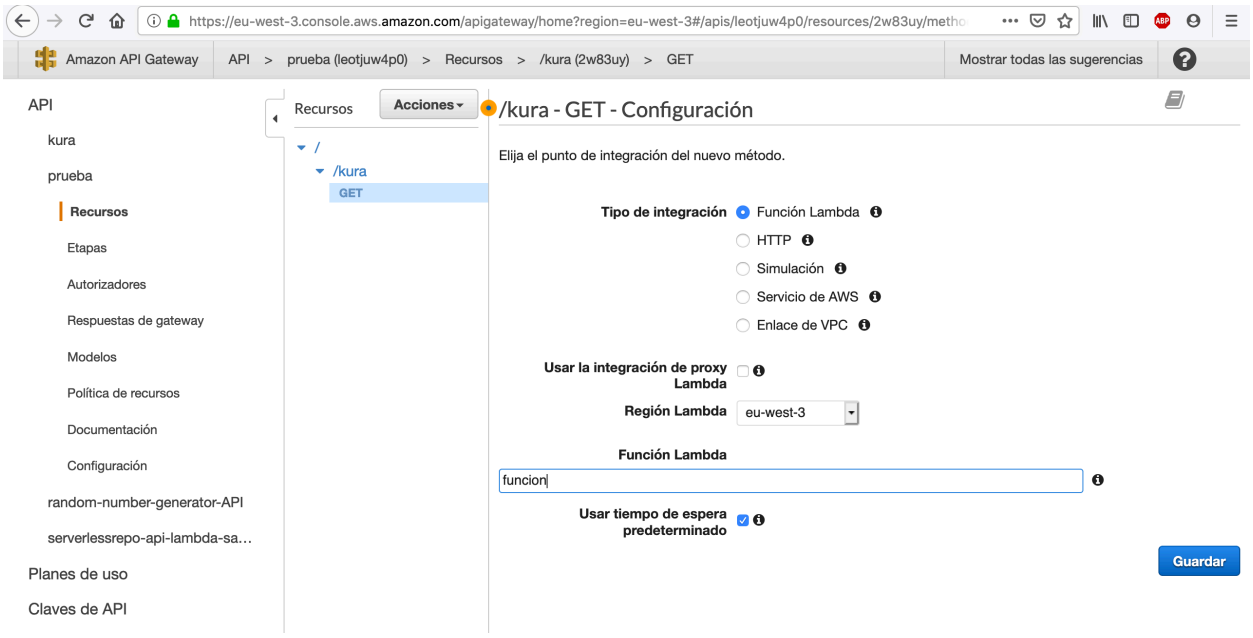


Figura 98: Creación método API

Se deberán aceptar los permisos que se soliciten cuando se haga click en Guardar. Como resultado, se mostrará la ventana que se muestra en la Figura 99: Esquema ejecución del método. En esta se representa un esquema de ejecución que tendrá la invocación de ese método.

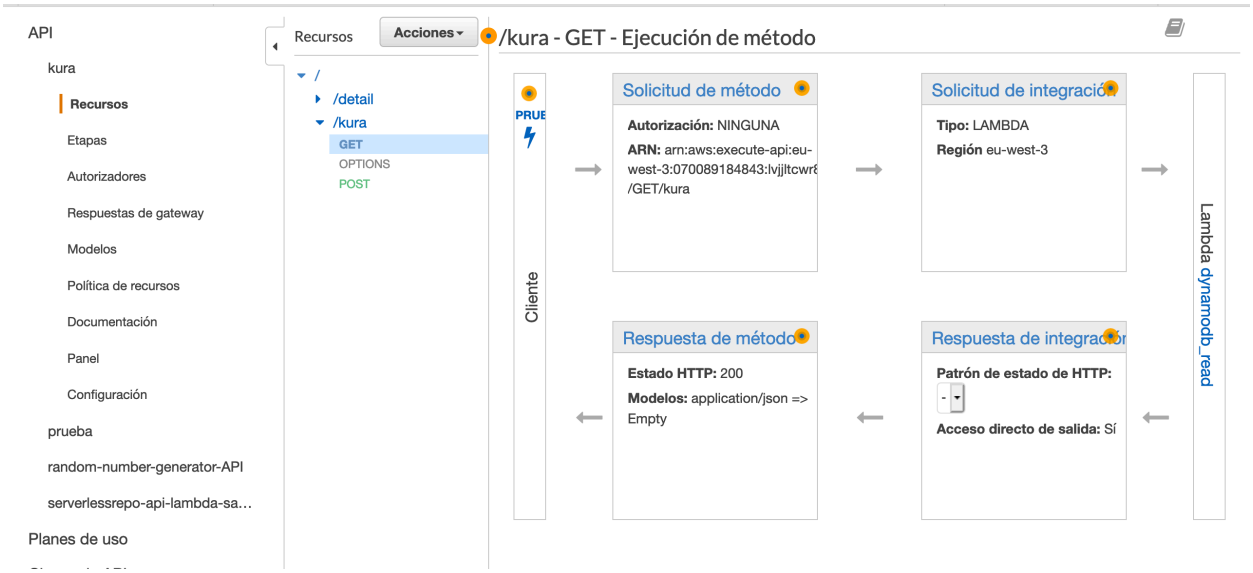


Figura 99: Esquema ejecución del método

Por último, será necesario establecer una plantilla de mapeo, para pasar a la función Lambda la información contenida en el cuerpo de la petición HTTP en formato JSON. Para ello, debemos hacer click en Solicitud de Integración -> Plantillas de mapeo -> Agregar plantilla de mapeo. En content type escribiremos application/json. A continuación, se deberán escribir las plantillas de mapeo. Las plantillas de mapeo de cada método se muestran en el apartado correspondiente de 5.4 API Gateway.

Tras haber guardado, debemos implementar la API. Para ello, hay que hacer click en Acciones -> Implementar la API. Crearemos una nueva etapa, con el nombre que queramos (en nuestro caso **prod**). Tras esto, se proporcionará la URL a la que se le tendrán que hacer las peticiones HTTP. Esto se muestra en la Figura 100: Implementación API.

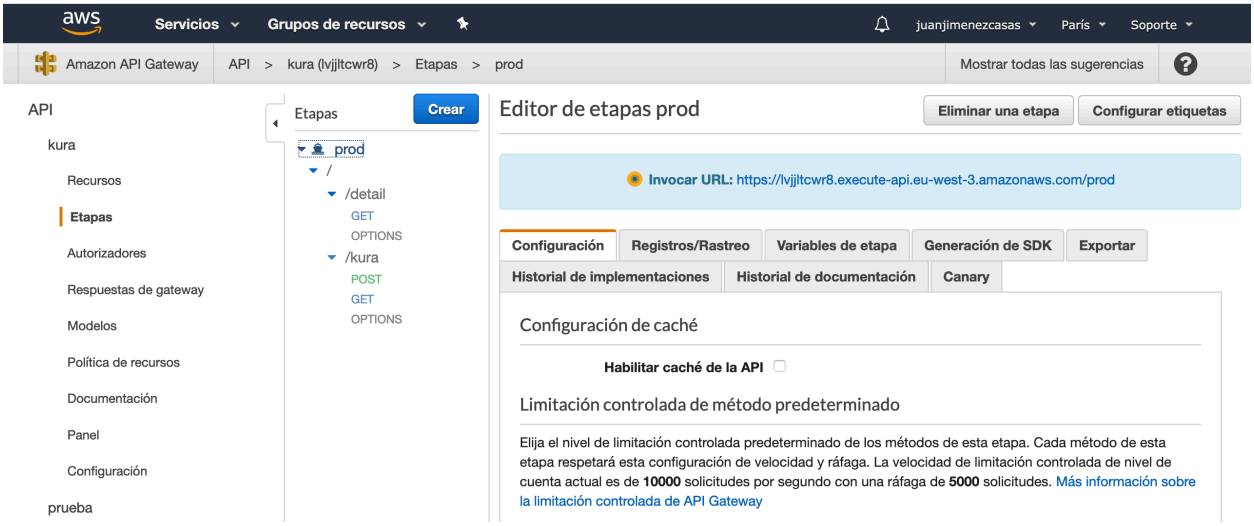


Figura 100: Implementación API

ANEXO D: DESPLIEGUE DEL SISTEMA

En este anexo se explican los comandos que debemos ejecutar para desplegar el sistema y que funcione tal y como se indica en esta memoria.

Para ello, es necesario haber creado los distintos componentes utilizados, como se indica en los anexos: Anexo A: Instalación de Sensores DHT11 y Leds en Raspberry Pi, Anexo B: Instalación y configuración de Eclipse Kura en Raspberry Pi y Anexo C: Creación de usuarios y configuración de servicios en AWS.

Además se deben haber creado las funciones, clases y bundles indicados en los apartados: 4 Aplicación desarrollada: Raspberry Pi y Eclipse Kura, 5 Aplicación desarrollada: Cloud Computing en Amazon AWS y 6 Aplicación desarrollada: Interfaz web Angular.

Una vez realizado todo esto, la puesta en marcha del sistema es relativamente sencilla.

7.8 Despliegue del bundle en Raspberry Pi

Una vez el bundle ha sido probado y todo funciona según lo esperado, este está listo para ser instalado en nuestra Raspberry Pi de forma permanente. Para ello se ha seguido el tutorial proporcionado por Eclipse Kura [26].

7.8.1 Creación Deployment Package

Un paquete de despliegue (Deployment Package) consiste en una serie de recursos agrupados en un único archivo de tipo paquete que puede ser desplegado en un *framework* OSGi a través del servicio `Deployment Admin` (administrador de despliegue).

En nuestro caso, el paquete de despliegue contendrá nuestro bundle únicamente y es una tarea muy simple generarlo:

- Creamos el JAR de nuestro bundle.
- Creamos un fichero de tipo `Deployment Package Definition` en la carpeta `resources/dp/` de nuestro proyecto.
- Indicamos en el fichero anterior los bundles que se van a empaquetar. En nuestro caso, solo el JAR de nuestro bundle.
- Guardamos los cambios en el fichero, hacemos clic derecho en él y seleccionamos `Quick Build`.

7.8.2 Despliegue en Raspberry Pi

Para desplegar el bundle en la Raspberry Pi de forma permanente, tenemos que:

1. Copiar el paquete de despliegue del bundle generado en el apartado anterior en el directorio /opt/eclipse/kura/kura/packages.
2. Editar el fichero /opt/eclipse/kura/kura/dpa.properties y añadir la línea:

```
heater=file\:/opt/eclipse/kura/kura/packages/heater.dp
```

Una vez hecho esto, en el próximo inicio de Kura el paquete se instalará y quedará en el dispositivo de forma permanente. De esta forma, se arrancará cuando encendamos la Raspberry Pi.

Si consultamos el fichero de registro /var/log/kura.log, que se muestra en la Figura 20: Fichero /var/log/kura.log, podemos observar cómo se publica la información de humedad y temperatura correctamente en el topic MQTT que hemos configurado con el CloudPublisher.

7.8.3 Activación o desactivación del bundle

Para activar o desactivar un bundle, debemos ejecutar los comandos proporcionados en la documentación de Eclipse Kura desde la Raspberry Pi [26]. Los comandos serían los siguientes:

- Nos conectamos al *framework* de OSGi:

```
telnet localhost 5002
```

- Mostramos la lísta de bundles que tenemos instalados.

```
ss
```

- Para activar o desactivar un bundle, debemos ejecutar el comando:

```
start ## / stop ##
```

Donde `##` indica el número del bundle obtenido al ejecutar el comando `ss`.

7.9 Cloud Computing en Amazon AWS

La puesta en marcha de los servicios implementados en Amazon AWS se hace de forma inmediata al crearlos. Es una de las principales ventajas del Cloud Computing, ya que no hay que preocuparse de esos aspectos. Sólo se consumirán recursos cuando se necesiten y estarán disponibles en todo momento. Para gestionarlos sólo tendremos que iniciar sesión en el sistema y acceder al recurso que queramos.

7.10 Aplicación Web en Angular

Para desplegar la aplicación web Angular, tenemos dos posibilidades. Una es utilizar la URL que proporciona este al crear una aplicación y la otra es descargar el código e implementar el servicio Web de forma local en nuestra máquina. En nuestro caso, usaremos la implementación de Stackblitz directamente.

7.10.1 Implementación con Stackblitz

Esta es la forma más cómoda, ya que automáticamente se realiza un control de versiones en GitHub, se instalan los paquetes necesarios de forma dinámica y además se facilita una URL para poder hacer uso de la aplicación desarrollada. Es la forma en la que la documentación de Angular [42] recomienda la implementación. Para ello, debemos crear un proyecto en Stackblitz. Debemos registrarnos con nuestra cuenta de GitHub para que se haga automáticamente un control de versiones. Tras estos pasos, tendremos nuestro entorno de desarrollo creado. Si hacemos click en el botón **Share**, se nos mostrarán diferentes URL. La URL para poder acceder a nuestra aplicación Web es la que se muestra en la Figura 101: IDE Stackblitz, en el campo APP URL.

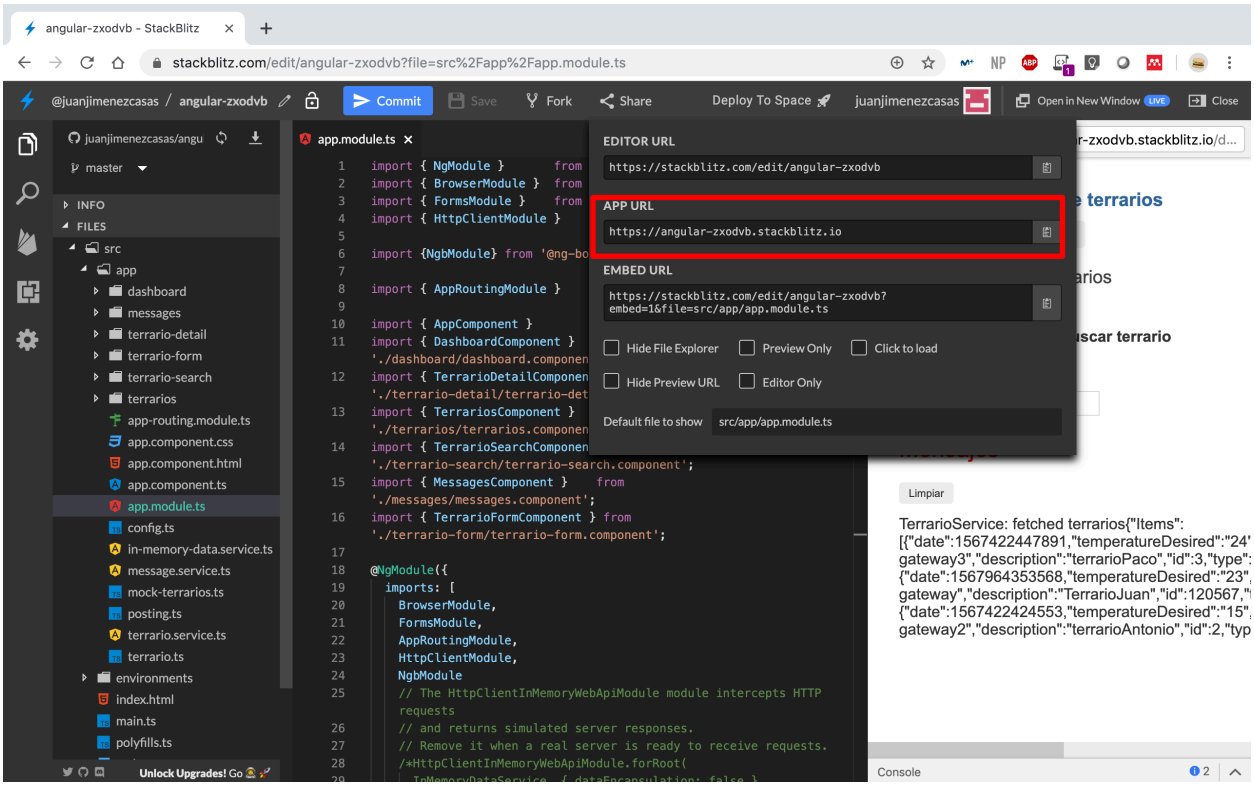


Figura 101: IDE Stackblitz

Si accedemos desde un navegador Web a la dirección https://angular-zxodvb.stackblitz.io, tendremos acceso a nuestra aplicación Web, como se muestra en la Figura 102: Aplicación Web.

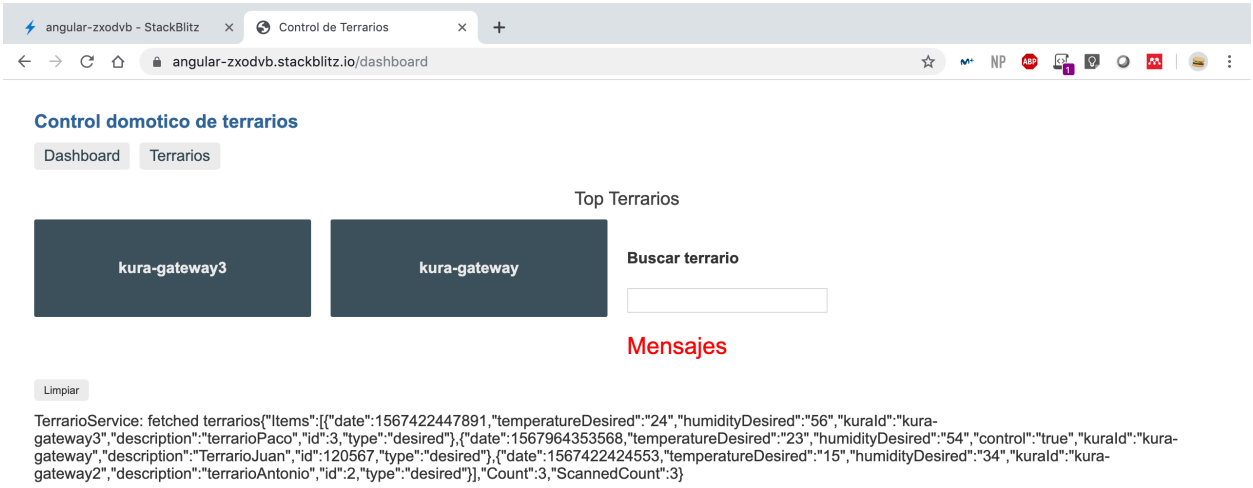


Figura 102: Aplicación Web

REFERENCIAS

[1] «Raspberry Pi 3B+ Specs and Benchmarks - The MagPi MagazineThe MagPi Magazine». [En línea]. Disponible en: https://www.raspberrypi.org/magpi/raspberry-pi-3bplus-specs-benchmarks/.

[2] «DHT11 Temperature and Humidity Sensor and the Raspberry Pi - Raspberry Pi Spy». [En línea]. Disponible en: https://www.raspberrypi-spy.co.uk/2017/09/dht11-temperature-and-humidity-sensor-raspberry-pi/#prettyPhoto.

[3] «Getting Started». [En línea]. Disponible en: https://eclipse.github.io/kura/dev/kura-setup.html.

[4] «Eclipse Kura | The Eclipse Foundation». [En línea]. Disponible en: https://www.eclipse.org/kura/.

[5] «Gateways | IoT development made simple - iot.eclipse.org». [En línea]. Disponible en: https://iot.eclipse.org/gateways/.

[6] «Gateways IoT | Aprendiendo Arduino». [En línea]. Disponible en: https://aprendiendoarduino.wordpress.com/2018/11/21/gateways-iot/.

[7] «What is OSGi? – OSGi™ Alliance». [En línea]. Disponible en: https://www.osgi.org/developer/what-is-osgi/.

[8] «What is OSGi (Open Service Gateway Initiative)? - Definition from WhatIs.com». [En línea]. Disponible en: https://searchnetworking.techtarget.com/definition/OSGi.

[9] «Benefits of Using OSGi – OSGi™ Alliance». [En línea]. Disponible en: https://www.osgi.org/developer/benefits-of-using-osgi/.

[10] «Architecture – OSGi™ Alliance». [En línea]. Disponible en: https://www.osgi.org/developer/architecture/.

[11] «Amazon Web Services (AWS): ¿qué es y qué ofrece?» [En línea]. Disponible en: https://www.ticportal.es/temas/cloud-computing/amazon-web-services.

[12] J. M. L. Sillero, «Amazon Web Services». [En línea]. Disponible en: https://www2.deloitte.com/es/es/pages/technology/articles/introduccion-a-amazon-web-services.html.

[13] «What is AWS IoT (Amazon Web Services internet of things)? - Definition from WhatIs.com». [En línea]. Disponible en: https://searchaws.techtarget.com/definition/AWS-IoT-Amazon-Web-Services-internet-of-things.

[14] «¿Qué es AWS IoT? - AWS IoT». [En línea]. Disponible en: https://docs.aws.amazon.com/es_es/iot/latest/developerguide/what-is-aws-iot.html.

[15] «¿Qué es Amazon DynamoDB? - Amazon DynamoDB». [En línea]. Disponible en: https://docs.aws.amazon.com/es_es/amazondynamodb/latest/developerguide/Introduction.html.

[16] «Cifrado en reposo de DynamoDB - Amazon DynamoDB». [En línea]. Disponible en: https://docs.aws.amazon.com/es_es/amazondynamodb/latest/developerguide/EncryptionAtRest.html.

[17] «What is Serverless? | Serverless Stack». [En línea]. Disponible en: https://serverless-stack.com/chapters/what-is-serverless.html.

[18] «¿Qué es AWS Lambda? - AWS Lambda». [En línea]. Disponible en: https://docs.aws.amazon.com/es_es/lambda/latest/dg/welcome.html.

[19] «Tiempos de ejecución de AWS Lambda. - AWS Lambda». [En línea]. Disponible en: https://docs.aws.amazon.com/es_es/lambda/latest/dg/lambda-runtimes.html.

[20] «▷ ¿Qué es una API REST? | Guía 【2019】 - Idento». [En línea]. Disponible en: https://www.idento.es/blog/desarrollo-web/que-es-una-api-rest/.

[21] «¿Qué es REST? Conoce su potencia | OpenWebinars». [En línea]. Disponible en: https://openwebinars.net/blog/que-es-rest-conoce-su-potencia/.

[22] «TypeScript - Wikipedia, la enciclopedia libre». [En línea]. Disponible en: https://es.wikipedia.org/wiki/TypeScript.

[23] «Qué es Angular | OpenWebinars». [En línea]. Disponible en: https://openwebinars.net/blog/que-es-angular/.

[24] «Overview». [En línea]. Disponible en: https://eclipse.github.io/kura/cloud-api/1-overview.html.
[25] «User guide». [En línea]. Disponible en: https://eclipse.github.io/kura/cloud-api/2-user-guide.html.
[26] Raspberry Pi Foundation, «Deploying Bundles». [En línea]. Disponible en: https://eclipse.github.io/kura/dev/deploying-bundles.html.
[27] «Configurable Application». [En línea]. Disponible en: https://eclipse.github.io/kura/dev/configurable_component.html.
[28] «Publishing Application». [En línea]. Disponible en: https://eclipse.github.io/kura/dev/heater_demo.html.
[29] «GPIO driver in Kura Wires». [En línea]. Disponible en: https://eclipse.github.io/kura/wires/kura-wires-gpio.html.
[30] «Políticas de AWS IoT - AWS IoT». [En línea]. Disponible en: https://docs.aws.amazon.com/es_es/iot/latest/developerguide/iot-policies.html.
[31] «API RESTful de sombra de dispositivo - AWS IoT». [En línea]. Disponible en: https://docs.aws.amazon.com/es_es/iot/latest/developerguide/device-shadow-rest-api.html.
[32] «Temas - AWS IoT». [En línea]. Disponible en: https://docs.aws.amazon.com/es_es/iot/latest/developerguide/topics.html.
[33] «¿Qué es AWS Lambda? | Serverless Stack». [En línea]. Disponible en: https://serverless-stack.com/chapters/es/what-is-aws-lambda.html.
[34] «API RESTful de sombra de dispositivo - AWS IoT». [En línea]. Disponible en: https://docs.aws.amazon.com/es_es/iot/latest/developerguide/device-shadow-rest-api.html.
[35] «Implementing a Serverless AWS IoT Backend with AWS Lambda and Amazon DynamoDB | AWS Compute Blog». [En línea]. Disponible en: https://aws.amazon.com/es/blogs/compute/implementing-a-serverless-aws-iot-backend-with-aws-lambda-and-amazon-dynamodb/.
[36] Raspberry Pi Foundation, «Installing operating system images - Raspberry Pi Documentation», 2008. [En línea]. Disponible en: https://www.raspberrypi.org/documentation/installation/installing-images/.
[37] Raspberry Pi Foundation, «Raspberry Pi- Quick start», 2012. [En línea]. Disponible en: https://eclipse.github.io/kura/intro/raspberry-pi-quick-start.html.
[38] Raspberry Pi Foundation, «Network Configuration». [En línea]. Disponible en: https://eclipse.github.io/kura/config/network-configuration.html.
[39] «Amazon AWS IoT™ platform». [En línea]. Disponible en: https://eclipse.github.io/kura/cloud-stacks/kura-aws-cloud.html.
[40] «Uso de políticas basadas en recursos para AWS Lambda - AWS Lambda». [En línea]. Disponible en: https://docs.aws.amazon.com/es_es/lambda/latest/dg/access-control-resource-based.html.
[41] «TUTORIAL: Desarrollar una API con integración HTTP que no sea de proxy - Amazon API Gateway». [En línea]. Disponible en: https://docs.aws.amazon.com/es_es/apigateway/latest/developerguide/api-gateway-create-api-step-by-step.html.
[42] «Angular - Architecture overview». [En línea]. Disponible en: https://angular.io/guide/architecture.